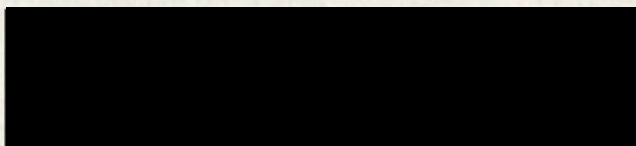
# Applications of Junction Theory

Stuart J. E. Baird

PhD Population Genetics
The University of Edinburgh
1994

In memory of Mrs A. L. Baird
and dedicated to Mrs O. J. Crawford

I composed this thesis, the work is my own



Stuart J. E. Baird

Contents

**(1) Introduction to Junctions**
1.0 Evolution of Many Loci ........ 2
1.1 Introducing Junction Theory ........ 7
1.2 The Junction Approach to Multilocus Problems ........ 9
1.3 Applications of Junctions in this Thesis ........ 12

**(2) The Mixing of Genotypes in Multilocus Clines**
2.0 Introduction ........ 15
2.1 Multilocus Clines ........ 16
2.2 The Junctions Model ........ 20
2.3 Comparing Analysis and Simulation ........ 22
2.4 Extending the Analysis ........ 28
2.5 Discussion ........ 33
2.6 Summary ........ 36
Appendix 2.1: The Fate of Introgressing Chromosomes ........ 38

**(3) Estimating the Age of Hybrid Zones**
3.0 Introduction ........ 45
3.2 Additive Selection ........ 48
3.2 Arbitrary Selection ........ 50
3.3 Modelling Finite Populations - a Preliminary Study ........ 54
  3.3.1 The Junctions Model ........ 56
  3.3.2 Results ........ 58
3.3 Discussion ........ 61

**(4) Invasion of Genomes**
4.0 Introduction ........ 64
4.1 Hitch-Hiking in a Single Population ........ 69
4.2 Hitch-Hiking in Spatially Structured Populations ........ 73
4.3 Discussion ........ 85

**(5) Contribution to Future Generations**
5.0 Introduction ........ 88
5.1 Branching Processes ........ 89
5.2 The Junctions Model ........ 92
5.3 Testing the Model ........ 95
  5.3.1 Drift ........ 95
  5.3.1 Recombination ........ 96
5.4 Discussion ........ 96

**(6) Discussion** ........ 102
ACKNOWLEDGEMENTS ........ 107
REFERENCES ........ 108

# (1) Introduction to Junctions

"Of course we shall eventually have to face up to multi-locus complexity."

(Dawkins 1982, p22)

1.0 Evolution of Many Loci ........ 2
1.1 Introducing Junction Theory ........ 7
1.2 The Junction Approach to Multilocus Problems ........ 9
1.3 Applications of Junctions in this Thesis ........ 12

# 1.0 Evolution of Many Loci

The evolutionary synthesis has been described as a period of mutual education between those who studied genetics, systematics, natural history and paleontology: When brought together, their different viewpoints led to a deeper understanding of one of the key concepts in modern thought. However, their perspectives remain very different, emphasising evolution at either the level of the organism or at the level of the gene. Explanations of the abundant variation that exists in natural populations, the processes involved in speciation, and the ubiquity of sexual reproduction cannot be given in terms of isolated genes, yet genes are the ultimate units of inheritance. To determine the level at which greatest understanding lies, we must ask to what degree sets of genes remain associated over time.

It is important to distinguish two levels at which genes at different loci may influence each other: Genes may *interact* functionally within an organism, as is usually the case with quantitative traits; and genes may be *associated* statistically within a population, so that a change in the frequency of one gene will be accompanied by changes in the frequencies of those genes with which it is associated. A precise understanding of how organisms work requires knowledge of gene interactions. A precise understanding of how organisms evolve requires knowledge of gene associations. Epistatic selection is the bridge

between these two levels: if functional interactions produce non-additive phenotypic effects (epistasis), selection will form associations between favourable sets of genes, making evolution of allele frequencies at the relevant loci interdependent. However, drift and migration are other sources of association between alleles at different loci. Thus, the need to consider many loci depends on the general importance of such associations in nature, rather than epistasis alone.

Association between alleles at different loci has been described in terms of the covariance in their allelic values or, equivalently for two loci, their pairwise linkage disequilibrium: a measure of the statistical association of alleles in forming gametes. For allele frequencies at a locus to evolve independently of the genetic background, that locus must be in linkage equilibrium with all others. Most classical population genetics assumes (for simplicity) that loci evolve independently, as recognised by Dawkins in the opening quote of this chapter. Yet at the most basic level, if we accept that sex and recombination are adaptive, associations between loci *must* have a central place in evolutionary theory, for without linkage disequilibrium recombination cannot affect the genetic composition of a population (Felsenstein, 1988).

Futuyma (1994) summarises the historical division arising from the issues of epistatic and pleiotropic effects. Theorists such as Fisher and Haldane placed little emphasis on gene interactions, and considered genes to have effects independent of their genetic background. In contrast workers such as Wright, Dobzhansky and

Mayr considered gene interactions central to evolution: The intricate construction and development of organisms requires a myriad of processes, each dependent on numerous genes. Substitution at one locus would therefore effect evolution at many others, as complex epistatic selection would tend to form associations between favoured collections of genes.

The build up of linkage disequilibria under simple models of strong epistatic selection is well studied: Franklin and Lewontin (1970) found that a 36-locus model with symmetric overdominance within loci, and multiplicative selection against homozygotes between loci, coalesced into a structure characterised by two complementary gametic types at high frequency, and almost complete linkage disequilibrium, leading Lewontin (1974) to propose that a new theory of population genetics was necessary, with the chromosome as the unit of selection. Clegg et. al (Clegg 1978; Clegg et al. 1980) observed no such persistent effects in *Drosophila* cage populations artificially started in total linkage disequilibrium. However their findings agreed with Franklin and Lewontin's view that two locus theory seriously underestimates the importance of linkage disequilibrium among pairs of loci embedded in a background of selected loci: neutral markers returned to linkage equilibrium in cage populations almost twice as fast as the two locus expectation, seemingly being 'pulled' to equilibrium by the cumulative effect of their genetic background. The breakdown of the artificially induced disequilibria was much more closely modelled by a computer simulation of 93 loci evenly spaced on a

single chromosome, with symmetric overdominance returning neutral genes to intermediate frequencies more effectively than a classical dominance model.

Sampling drift creates transient linkage disequilibria which can cause a net reduction in a population's response to selection (Hill & Robertson, 1966), and lead to hitch-hiking during the selective increase of an initially rare allele (See Kreitman 1987 for review). In theory this could significantly reduce gametic diversity. For example genes hitchhiking with selected loci seem to have led to reduced polymorphism on the fourth chromosome of *Drosophila* (Berry, Ajioka, & Kreitman, 1991).

A more general role of linkage disequilibria derives from Kondrashov's deterministic mutation hypothesis for the advantage of sexual reproduction (See Kondrashov 1988 for review). Recombination increases the variance in genomic 'contamination' by mutations, occasionally bringing a number of them together. Epistatic selection can remove such collections efficiently as the marginal selection on additional mutations becomes large, reducing the mutation load and creating negative linkage disequilibrium between mutations. It follows that, if Kondrashov's hypothesis is correct, sexual populations would possess very large numbers of loci maintained in slight linkage disequilibrium.

Bulmer (1980) has shown that the most immediate effect of stabilising selection on a polygenic character is the creation of

associations between the loci involved, the cumulative effect of many pairs in slight linkage disequilibrium being large enough to change the phenotypic variance considerably, before there is any appreciable change in gene frequencies.

The process most likely to produce strong linkage disequilibria is migration (Li & Nei, 1974; Slatkin, 1975). With exchange of a proportion m of individuals between discrete demes, the pre-existing associations between the loci of individuals derived from different demes are broken down by recombination (r), giving interactions of order m/r. In a cline, linkage disequilibrium is generated in proportion to the gradients in allele frequencies. At the centre of the cline the gradient is determined by the strength of selection, producing associations of order s/r. Barton (1983) argued that for this reason multilocus interactions could be important in hybrid zones, where migration brings together chromosomes which may differ at many loci.

Given the importance of multilocus effects, and the blossoming database of multiple locus information made possible by advances in molecular biology, it is important to develop methods of mathematical and computational analysis to help us to cope with the complexity of many loci.

## 1.1 Introducing Junction Theory

Junction theory is a tool for describing the mixture of genetic material resulting from recombination. In order to analyse the loss of variance due to inbreeding Fisher (1949;1953) developed a representation of continuous genetic material in terms of the 'junctions' along its length where material of different ancestry has come together as a result of recombination. A new junction is formed when a crossover occurs in a region for which the parent organism is heterozygous. Once produced, junctions are inherited like point mutations, and may be lost or instead may increase to fixation. Knowing the frequency of all haplotypes and the rate of production of junctions, Fisher calculated the expected length of heterogenic material for systems such as sib-sib mating. The production of junctions between two original haplotypes illustrates how complex offspring can be represented in a very elegant way (Figure 1.1).

Robertson (1977) used such a simulation to extend his work on artificial selection to the infinite locus case. He recorded the physical makeup of gametes in terms of the origin and 'breakpoints' of their constituent blocks and combined this with an elegant mechanism for the exact calculation of the variance associated with each newly produced block to simulate selection on an infinite number of loci. More recently Franklin (1977) and Stam (1980) have used this simulation approach during further investigations into the effects of inbreeding. Barton's (1983)

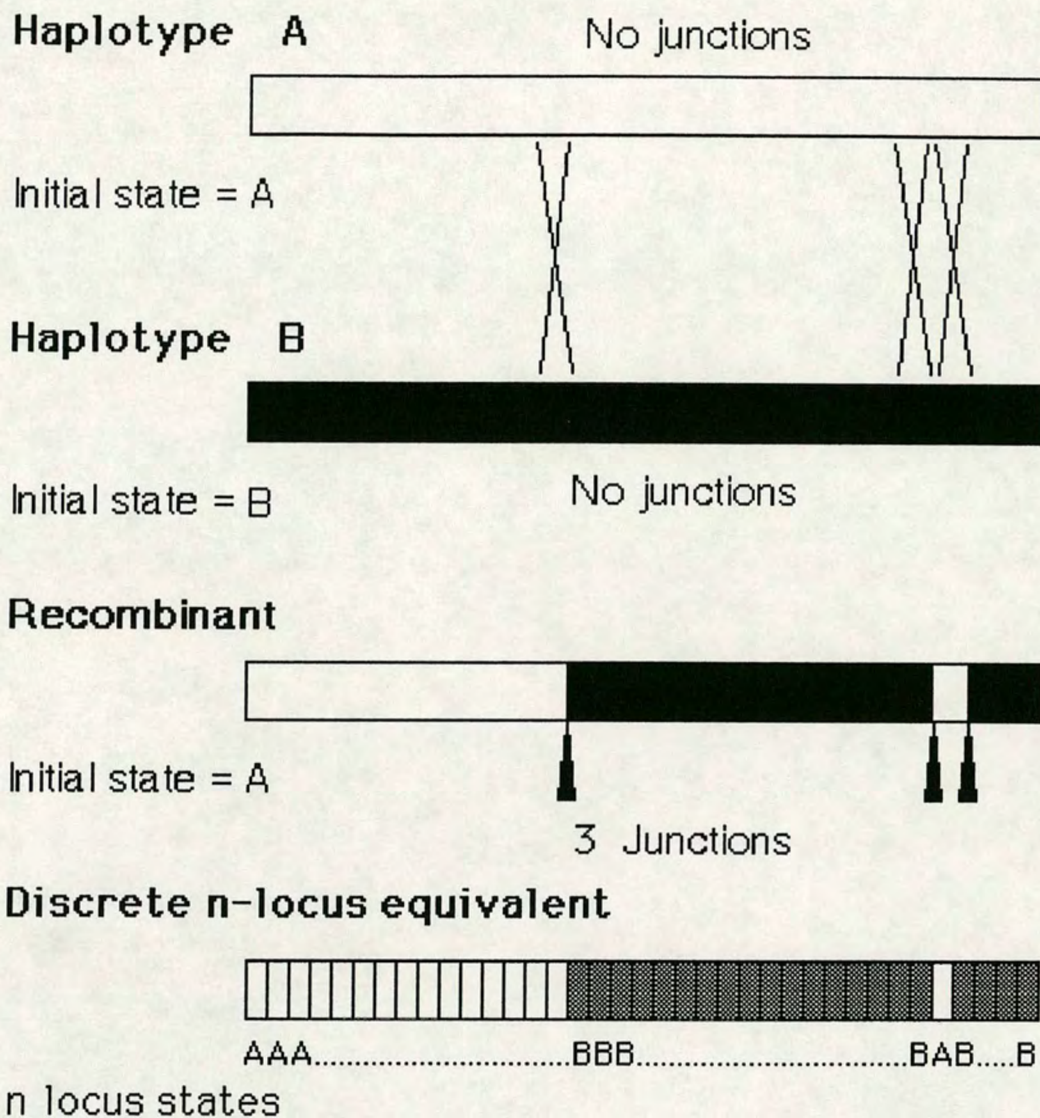



Figure 1.1: The Junctions Approach

analytical work on multilocus clines, which is the starting point for chapter 2, considers the break-up of 'blocks' of homogenous genetic material by recombination.

Though different terminology is used in these studies, the representations of genetic material are equivalent, and the descriptive convenience of 'blocks' of genetic material meeting at 'junctions' will used throughout the following chapters.

## 1.2 The Junction Approach to Multilocus Problems

We have seen why many important evolutionary problems can only be understood by considering many loci. The junction approach simplifies analysis and simulation of multilocus processes involving populations which can initially be described in terms of a finite number of distinct haplotypes.

We have seen the illustration of how junctions can be used to describe the mixture of two haplotypes (Figure 1.1). More generally a population of 2N haploids, with C distinct haplotypes of L discrete loci, can be fully described by the set C of haplotypes and a record of the 2N starting states of the haploids, where a starting state corresponds to one of the set C. Recombination tends to make the description of a haploid more complex, increasing the amount of information we need for a complete description of the population. In the worst case after recombination between haplotypes of map length R, which differ

throughout their length, a daughter haploid is described by its starting state and on average R junctions. Associated with each junction we imagine a variable indicating the next state, again one of the set C. Using the junction approach the greatest number of variables we need to follow at time t in the absence of selection is approximately:

$$CL + 4NRt \qquad (1.1)$$

If we initially have highly ordered haplotypes, which are uniform over all loci, or the effect of which can be expressed by some smooth function, then we no longer need to consider all points on each haplotype separately and (1.1) reduces to:

$$4NRt \qquad (1.2)$$

If we initially have only 2 highly ordered haplotypes, which are uniform over all loci then we no longer need to indicate the next state after a junction, as by definition it must be the alternative to the state before the junction, and (1.2) reduces to:

$$2NRt \qquad (1.3)$$

The conventional approach of following the state at each locus of each haploid requires 2NL variables at all times. In the case of (1.3) the advantage of junction theory is at least L/Rt fold at time t: the advantage of junction theory is directly proportional to the number of loci we wish to consider. For many applications the number of loci is no longer a cost factor at all, so we can model continuous genetic material with an effectively infinite number of loci or base pairs!

The argument in favour of junctions has so far been based on the accumulation of junctions by recombination, without reference to

selection or drift. Drift in a finite population will tend to reduce the rate of production of junctions over time until their numbers equilibrate at 2NJ, where the population has fixed for some recombinant which has J junctions. Selection for one particular haplotype will have a similar, though faster, effect. In general, epistasis between loci will reduce the number of junctions in the population. For example, Chapter 2 considers heterozygote disadvantage which favours two parental haplotypes over their recombinant descendants. The worst case for the accumulation of junctions under selection is heterosis such as was observed in Clegg's study (Clegg et al., 1980), where the fittest haplotype does not exist in the base population, and is instead created through recombination.

Perhaps the elegance of the junctions approach arises because, like many of the best ideas, it copies the actions of nature. Indeed, if we had two sequenced homologous divergent chromosomes, we should be able to simply note junctions by inspection of any recombinant daughter chromosome. Work of this nature has already been done on prokaryotes (Maynard Smith. 1990 for review), revealing the mosaic structure of genes caused by both transformation events and recombination. Detecting recombination in eukaryote lineages requires more sophisticated analysis due to its high frequency relative to the rate of mutation (Sawyer 1989, Stevens 1985). Nevertheless, the rapid progress in sequencing technology seems likely to bring junctions increasingly to the forefront of all fields involving molecular genetics.

## 1.3 Applications of Junctions in this Thesis

The method of junctions will be applied to several problems in evolutionary biology- hybrid zones, invasion of new genomes, and ancestry. Other applications will be discussed later.

Hybrid zones are a window on the processes of evolution, informing us about the magnitude of selection and dispersal in natural populations, and the nature of speciation. Secondary contact between populations which have diverged in isolation will produce associations between many loci, and models considering only a few loci may be inappropriate. If divergence is at very many loci, each of small effect, then two distinct haplotypes will emerge, or more accurately, the distribution of haplotypes will become bimodal, tending to separate between the two sub-populations. If such populations are reunited then we can approximate the base population as having two distinct highly ordered haplotypes as in (1.3), and the junction approach proves highly efficient. Chapters 2 and 3 consider what happens when populations come together to form a stable hybrid zone.

Instead of forming a stable hybrid zone, genes may spread through the range of a population. Chapter 4 considers what happens when favoured genes invade the genome of an established population.

Tracing the ancestry of human populations has produced great popular interest. More generally, analysis of ancestry may inform us about cladogenesis and the nature of macroevolution. Recombination turns ancestral trees into nets, greatly complicating analysis of lineages. Chapter 5 uses junctions to consider the genetic contribution to future generations of individuals in finite populations in the absence of selection: Each individual's genome is marked as distinct from all others, and then followed as it is broken up by recombination and lost by drift. Again the junction approach is highly efficient, corresponding to (1.2).

# (2)
# The Mixing of Genotypes in Multilocus Clines

THE MATERIAL IN THIS CHAPTER IS THE BASIS OF A PAPER ENTITLED "A SIMULATION STUDY OF MULTILOCUS CLINES". *EVOLUTION* IN PRESS.

APPENDIX 2.1 WAS COMPOSED BY N. H. BARTON

2.0 Introduction ..... 15
2.1 Multilocus Clines ..... 16
2.2 The Junctions Model ..... 19
2.3 Comparing Analysis and Simulation ..... 22
2.4 Extending the Analysis ..... 28
2.5 Discussion ..... 34
2.6 Summary ..... 36
Appendix 2.1: The Fate of Introgressing Chromosomes ..... 38

# 2.0 Introduction

Hybrid zones are common in nature, and their implications for the nature of species boundaries make them a rich area for research (see Barton and Hewitt 1985; 1989 reviews). For example, the toads *Bombina bombina* and *Bombina variegata* differ appreciably in appearance, behaviour, and preferred micro-habitat, and are known to differ at many electrophoretic loci, yet they hybridise in a long, narrow zone of hybridisation which runs around Eastern Europe (Szymura & Barton, 1986). The theory of clines maintained by a balance between selection and dispersal allows us to estimate the number of loci at which hybridising taxa such as *Bombina* differ (Szymura & Barton, 1991). Migration of individuals toward the centre of a hybrid zone, carrying alleles peculiar to one or other of the two taxa, will create associations between these alleles near the centre of the zone. Selection against hybrid offspring may also contribute to linkage disequilibrium. Genes which cross the centre of the zone will be subject to strong selection, as they will tend to be associated with other invading genes. Clines for selected loci will therefore be pulled together, strengthening each other to give a coincident set of steep clines which may have a sharp central step. Examples include *Bombina,* the cottonwood trees *Populus* sp. (Paige, Capman, & Jennetten, 1991), the European house mouse *Mus* (Hunt & Selander, 1973), the Alpine grasshopper *Podisma pedestris* (Barton & Hewitt, 1981), the Australian

grasshopper *Caledia captiva* (Moran, Wilkinson, & Shaw, 1980); See Harrison (1993) for reviews.

The maintenance of linkage disequilibria in hybrid zones prompted Barton (1983) to analyse the degree to which it causes loci to 'co-operate'. He found a critical level of selection above which loci no longer act independently: under weaker selection very small homogeneous blocks of loci are dominant in the population at equilibrium, so that loci act independently. Barton's mathematical analysis is the basis for the work in this chapter. A general infinite-locus simulation model is developed from Fisher's junctions method. This model is used to assess the robustness of the Barton's conclusions, which apply at equilibrium for what is from necessity a very restrictive model.

## 2.1 Multilocus Clines

Barton (1983) analysed multilocus clines by considering how recombination breaks down the associations between alleles which are produced by migration, and how selection against hybrids counters this process by removing the products of recombination. In the following descriptions, upper case is used for variables which apply to discrete quantities, for example loci or chromosomes, and lower case for continuous variables applying to fractions of chromosomes or populations. In a departure from Barton's (1983) notation, L is used to denote the number of loci instead of n. This is to avoid confusion with N, the population size. Barton considers a pair of infinite diploid

populations differing at L evenly spaced loci, and which exchange a proportion m individuals each generation. Selection is by additive heterozygote disadvantage, with fitness (1-Ys), where Y is the number of heterozygous loci. This is only a good approximation for multiplicative selection $(1\text{-s})^{Y}$ for the case of weak overall selection (Ys<<1). As the populations are infinite, immigrants and recombinants will always recombine with pure native individuals, and immigrant alleles will therefore always be heterozygous. Thus on a chromosome with L loci, heterozygote disadvantage at Y of those loci is equivalent to a haplotype fitness (1-Ys), and selection on haplotypes can be used for simplicity: the total selective pressure over a chromosome is **S**=Ls. Recombination between adjacent loci at a rate r is assumed to give a total map length of **R**=(L-1)r . Recombination, like selection, is assumed very weak (**R**<<1) with a limit of one crossover per generation so that recombinant chromosomes have at most one block of Y contiguous immigrant alleles, and interference can be ignored. The multiple blocks likely under stronger recombination make Barton's analytical model intractible. Because the present study is centred on the critical value of selection, we will consider only the case of an infinite number of loci, as this is the case for which Barton found the critical value of selection to be most obvious.

In the limit of infinite loci, Barton considered the continuous variable y=Y/L, the proportion of the chromosome which carries immigrant alleles. The rate of increase $\frac{dp}{dt}$ (y) of blocks size y, with frequency spectrum p(y) was found to be:

$$\frac{dp(y)}{dt} = -(S+R)yp(y) + 2R\int_y^1 p(z)dz + m\delta(1-y) \qquad (2.1)$$

(Barton 1983, Eq. 4)

where $\lim_{x\to 0}\int_0^x \delta(x)dx = 1$, and $\delta(x)$ is the Dirac delta function.

Here, the first term expresses the loss of blocks of size y that are selected, or broken up by recombination, the second term expresses the production of blocks size y by the break-up of larger blocks up to and including unbroken chromosomes (size 1), and the third expresses the spike of unbroken chromosomes due to immigration. It is important to clarify the distinction between the *frequency* of blocks introduced, and the *frequency spectrum*. In an infinite population invading blocks of all sizes must by definition have frequencies in (0,1) tending to zero. However, the frequency spectrum of blocks in $(0,\infty)$ allows us to compare the proportion of the infinite population made up of these different sizes of block. The equilibrium solution for the frequency spectrum of blocks size y is then:

$$p_{eq}(y) = \frac{m\theta}{S(1+\theta)}\left\{\delta(1-y) + \frac{2}{(1+\theta)}\, y^{-\left(\frac{3+\theta}{1+\theta}\right)}\right\} \qquad (2.2)$$

(Barton 1983, Eq. 5)

Here, $\theta$ is the ratio of selection to recombination S/R, termed the coupling coefficient. The average over the frequency spectrum of immigrant alleles is:

$$\bar{p} = \int_0^1 y p(y)\, dy = \begin{cases} \dfrac{m\theta}{S(\theta - 1)} & (\theta \geq 1) \\ \infty & (\theta \leq 1) \end{cases} \quad (2.3)$$

(Barton 1983, Eq. 6a)

$$\therefore \quad s^*/S = m/(S\bar{p}) = \begin{cases} (1-1/\theta) & (\theta \geq 1) \\ 0 & (\theta \leq 1) \end{cases} \quad (2.4)$$

(Barton 1983, Eq. 6b)

$s^*$ is the effective selection coefficient, that is the equivalent selection necessary to maintain a single allele at frequency $\bar{p}$ under a migration selection balance. The ratio $s^*/S$ expresses the proportion of the genome which would have to act in association under selection to maintain a given allele frequency. When selection is stronger than recombination ($\theta > 1$), the proportion of immigrant alleles carried on fragments of length y rises slower than $y^{-1}$ as $y \to 0$; because $\int y^{-b}\, dy$ is finite for $b > 1$ selection can hold introgression down to a finite level. However, when selection is weaker than recombination ($\theta < 1$), the proportion of alleles on small blocks rises faster than $y^{-1}$, so that these fragments make up in aggregate an overwhelming proportion of the population; the allele frequency rises indefinitely, and the effective selection is zero. Thus, there are two distinct domains, sharply separated at the critical value $\theta_c = 1$; when coupling is strong, the chromosome acts as the "unit of selection" ($s^* \approx S$), while when coupling is weak, selection does not act coherently, and $s^* << S$.

## 2.2 The Junctions Model

I will describe the process of creating simulations of the above situation in two stages: firstly the design of a model which can address the problem, and secondly the recasting of the relevant variables in terms of directly measurable quantities to allow comparison of simulation results to the previous analysis. Modelling selection, recombination, and migration is relatively straightforward. However, simulating an infinite population of chromosomes, each having an infinite number of loci, presents unusual problems.

If we consider secondary contact between two populations which have diverged at many loci, then we can designate chromosomes from one population as type A, and from the other type B, just as in Figure 1.1. The proportion of immigrant alleles y on a recombinant chromosome can easily be calculated from the position of junctions along its length. Given that the populations are infinite, the fitness of an individual is then (1-yS) where **S** is the total selection on a complete immigrant chromosome, as in Barton's model. Similarly recombination is uniform along the chromosome, with a number of chiasmata drawn from a Poisson distribution with expectation **R,** where **R** is the total map length and assuming no interference (See Appendix 1,1.0 for discussion). Distributing chiasmata in this way relaxes the weak recombination restriction of the analytical model, giving the possibility of multiple crossovers in one individual in the simulations.

The analytical model considers exchange between two demes, each of which has an infinite number of individuals, so that blocks of immigrant alleles will be rare in each deme. We can then assume that two individuals carrying immigrant alleles will never meet, but instead always reproduce with native individuals. Likewise, individuals dispersing from the deme will almost always have the pure native genotype. This has two useful consequences: firstly by symmetry we need only consider one of a pair of such demes which exchange equal numbers of individuals. Secondly, we can simulate an infinite population exchanging a finite number $M_o$ of immigrants each generation simply by following the increasing *but finite* number J of individuals which possess any amount of immigrant material. The analytical model considers an infinite population exchanging a proportion m of its individuals. A proportion of infinity is also infinity, which is rather impractical to simulate, so instead a finite number of individuals are exchanged. This introduces a source of sampling drift, but is perhaps more intuitively satisfying.

A second departure of the simulations from the analytical model is that generations are discrete. This means that under strong selection the order of events: migration, selection, then recombination becomes significant. Although a finite number $M_o$ of individuals invade the population each generation, these are subject immediately to selection, so the effective number of immigrants M, for the purposes of later analysis, is $(1\text{-}S)M_o$. The

first generation of individuals is created by the introduction of $M_0$ pure immigrant individuals. Data on the start state and junctions (See Fig. 1.1) of every chromosome are stored in each generation. Parents are drawn from the population at random, and the number of offspring from each pairing is drawn from a Poisson distribution with expectation twice the average fitness of the parents, such that if $\mathbf{S}=0$, reproduction will leave the ammount of immigrant genetic material in the population unchanged. As the native population is assumed to be infinite, each offspring is automatically given one pure native parent. The positions of all chiasmata for a chromosome pairing are drawn, ordered and then applied to the lists of junctions of the parents, to produce a single offspring which, if it contains any immigrant genetic material, is added to the stored individuals of the next generation.

## 2.3 Comparing Analysis and Simulation

Barton's analytical results can be recast in terms of the finite measurable quantities of the simulation model $\mathbf{S}$, $\mathbf{R}$, M, J, and $\overline{p}_J$, the average frequency of introgressing alleles in the J individuals which contain some immigrant material. It is obviously impossible in practice to calculate frequencies within an infinite population. However we can multiply frequencies p(y) and m by the population size to work in terms of the absolute number $\Psi(y)$ of blocks size y in the population, and the absolute number of

immigrants M. Thus $\Psi(y)=Np(y)$, and $M=Nm$. Then, for $N\rightarrow\infty$, and $m\rightarrow 0$, Eqs. 2.1 and 2.2 are equivalent to, respectively:

$$\frac{d\Psi(y)}{dt} = -(S+R)y\Psi(y) + 2R\int_y^1 \Psi(z)dz + M\delta(1-y) \qquad (2.5)$$

and

$$\Psi_{eq}(y) = \frac{M\theta}{S(1+\theta)} \left\{ \delta(1-y) + \frac{2}{(1+\theta)} y^{-\left(\frac{3+\theta}{1+\theta}\right)} \right\} \qquad (2.6)$$

Using the same reasoning we can derive an expression equivalent to Eq. (2.3), for the average frequency of immigrant alleles $\bar{p}_J$ over the J individuals followed in the simulations. By definition only the J individuals have immigrant blocks, so $\Psi(y) \equiv \Psi_J(y)$, where $\Psi_J$ is the absolute number of blocks in the J individuals. It follows

$\bar{p} = 1/N \int_0^1 y\Psi_J(y)dy$, and $\bar{p}_J = 1/J \int_0^1 y\Psi_J(y)dy$, so $\bar{p}_J = N\bar{p}/J$. Since $M=Nm$, it follows for $N\rightarrow\infty$ from Eq. (2.3):

$$\bar{p}_J = \begin{cases} \dfrac{M\theta}{JS(\theta - 1)} & (\theta \geq 1) \\ \infty & (\theta \leq 1) \end{cases} \qquad (2.7)$$

and from Eq. (4):

$$s^*/S = M/(JS\bar{p}_J) = \begin{cases} (1-1/\theta) & (\theta \geq 1) \\ 0 & (\theta \leq 1) \end{cases} \qquad (2.8)$$

Given $M=Nm$, and $N \rightarrow \infty$ in the analysis, we would expect that for large $M/m$ the simulation results should closely approximate these equations. For the purposes of simulation, the total map length R of chromosomes was held constant at 0.15. This is perhaps more realistic for a portion of a chromosome; however the value was chosen for convenience, as Barton's analytical results indicated that $s^*/S$ should depend solely on the coupling coefficient $\theta = S/R$, though with the assumption that S and R are weak. The total selection pressure S was varied between 0.0375 and 0.6, giving a range of $\theta$ from 1/4 to 4. Simulations were run in groups of ten replicates for 2500 generations from initial contact, with censuses every ten generations. Less frequently, the distribution of blocks in the simulation population was estimated by counting the number of blocks in different size classes. The breadth of each class was decreased exponentially allowing fine resolution for smaller blocks. A block is assigned to size class k if it is less than or equal to the $k^{th}$ class boundary, but greater than the $(k+1)^{th}$, where the $k^{th}$ boundary is $(7/8)^k$, and $40<=k<=0$. Thus size class zero runs from 1 to 7/8, and so on. The number of blocks in a particular size class is then used as an estimator for the density of blocks at the mid-point of that class on a log scale. Figure 2.1 shows the effective selection on a locus as a ratio of the total selection acting on the chromosome ($s^*/S$), for both the simulations (calculated as $s^*/S = M/SJ\bar{p}_J$), and the equilibrium solution for $t = \infty$; (Eq. 2.7). This ratio is equivalent to the proportion of the genome which would have to act together under selection to maintain the observed allele frequencies.

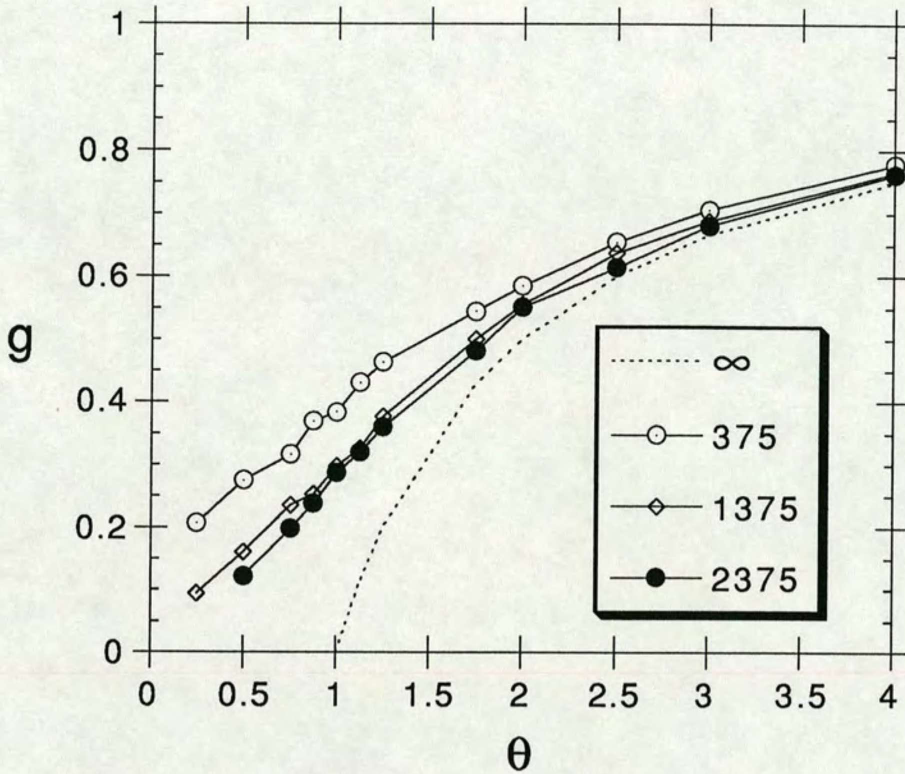



Figure 2.1: The effective proportion of the genome acting together (g), for different values of coupling (θ). Plots are shown for the equilibrium solution (∞), together with simulation results after 375, 1375, and 2375 generations, to indicate the rate of change. Simulation results are time averaged over 250 generations within runs, time averages are averaged over 10 replicate runs.

The simulation and equilibrium results converge for strong coupling. However, as coupling becomes weaker they diverge substantially. The critical value of selection found at $\theta = 1$ for equilibrium is not apparent even after more than 2000 generations of simulation. We would expect the difference between the simulations and equilibrium might be greatest for weak selection, as the population must progress much further from its initial state of pure native chromosomes to reach the high entropy of independently acting loci. Strong selection will tend to maintain the strong disequilibrium characteristic of the initial pure population, and therefore equilibrium should be reached more rapidly. Some further explanation is necessary however, to explain the absence of any marked transition point from independence to interaction of loci.

As mentioned in relation to Eq. 2.4, the behaviour predicted for $\theta < 1$ depends on small blocks of immigrant alleles dominating the population. Figure 2.2 compares the distribution of blocks observed in the simulation population after 100 and 2500 generations to the expected distribution at equilibrium (Eq. 2.5) under the extremes of strong and weak selection. The analytical and simulation results coincide precisely over a large range of block sizes: from 0.5 down to 0.001 for strong coupling, and 0.01 for weak coupling. The peak of large blocks around size 1 found in the simulations is due to pure immigrant chromosomes entering the population.

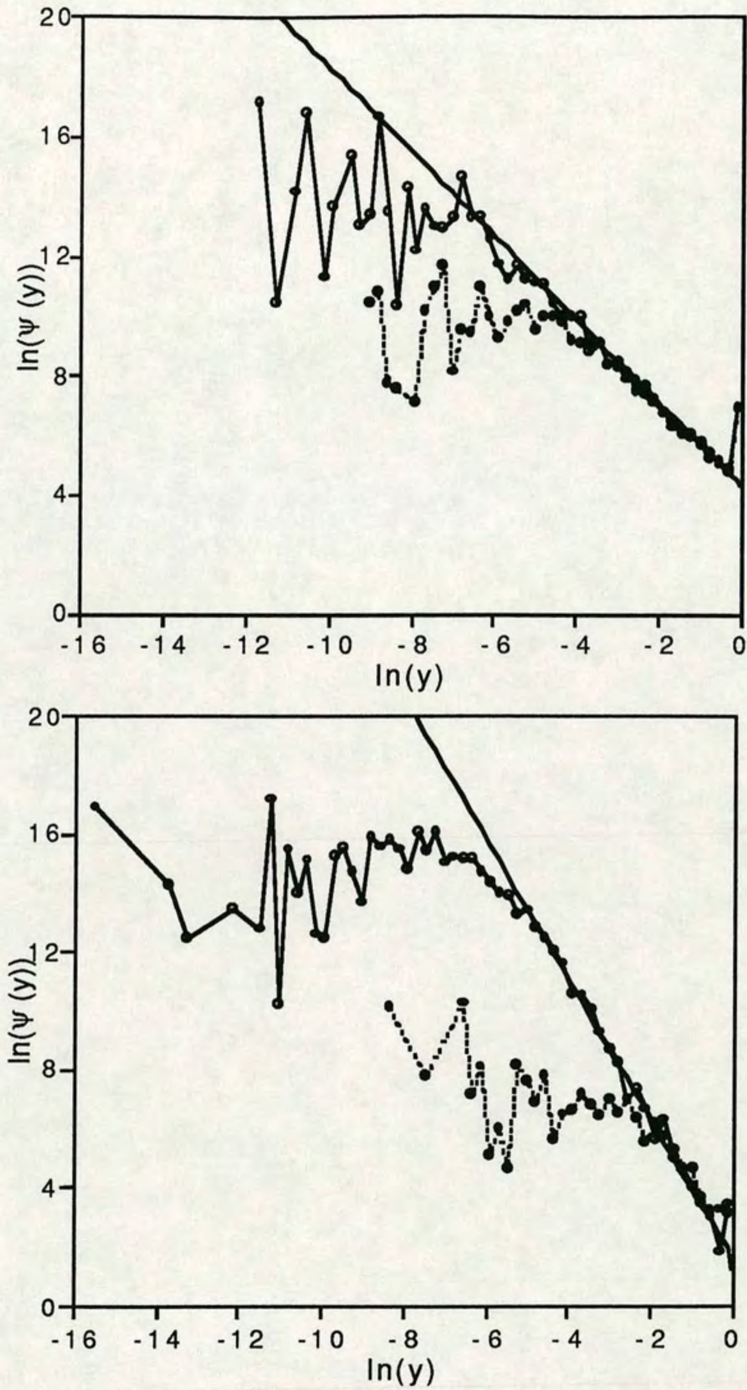



Figure 2.2: Comparison of observed block distributions to equilibrium prediction for $\theta = 4$ (above), and $\theta = 0.5$ (below). Filled circles: The observed distribution of block sizes found after 100 generations. Open circles: The observed distribution of block sizes found after 2500 generations. Bold line: The equilibrium solution for the distribution of block sizes (from Eq. 2.5). Simulation results are averages over ten replicates.

## 2.4 Extending the Analysis

With the above results in mind Barton has derived an exact solution for the frequency of blocks at time t, under additive selection (appendix 2.1). As before this can be recast in terms of the absolute number of blocks for comparison with the simulation results.

An alternative approximation for the frequency of blocks at time t can be solved numerically for any form of selection: Eq. 2.5 quantifies the change in number of blocks size y in terms of the creation of new blocks by recombination of the existing larger blocks, and their removal at a rate depending on y. We can summarise the terms of Eq. 2.5 by referring to the creation function $\alpha(y,t)$, and the removal function $\Omega(y)$:

$$\frac{d\Psi(y)}{dt} = -\Omega(y)\Psi(y) + \alpha(y,t) \qquad (2.9)$$

In the case of additive fitness $\Omega(y)=(S+R)y$†, or for an arbitrary selection function $S(y)$, $\Omega(y)=S(y)+Ry$ . For all forms of selection $\alpha(y,t)=2R\int_y^1 \Psi(z,t)dz + 2RM$, i.e. blocks are created by recombination over all blocks larger than y. Iterating the

† $\Omega=(S+R)y-SRy^2$ for large S and R. This correction makes no significant difference to later calculations.

discrete-time version of Eq. 2.8, for comparison with the simulation results, from $\Psi(y)=0$ we derive an expression $\Psi(y,t)$ for the number of blocks of size y after t generations:

$$\Psi(y,t) = \sum_{\tau=0}^{t-1}(1-\Omega(y))^{\tau}.\ \alpha(y,(t-1)-\tau) \qquad (2.10)$$

Unfortunately this equation is self referential as we need to know the numbers of existing blocks $\Psi(y,t)$ to calculate the rate of creation of new blocks $\alpha(y,t)$. However, we do know the maximum numbers that blocks can achieve, provided they are censused after selection, that is their numbers at equilibrium, $\Psi_{eq(y)}$. The blocks larger than y will never exceed their equilibrium numbers, and therefore the rate of creation of blocks at any time t will never be greater than:

$$\alpha_{max}(y,t)=2R\int_{y}^{1}\Psi_{eq}(z)dz + 2RM \qquad (2.11)$$

Iterating the discrete-time version of Eq. 2.8 from $\Psi(y)=0$, and assuming $\alpha(y,t) = \alpha_{max}(y,t)$, we therefore derive an upper bound U(y,t) on the number of blocks of size y after t generations:

$$U(y,t) = \left( \sum_{\tau=0}^{t-1} (1-\Omega(y))^{\tau} \right) . \, \alpha_{max}(y,t) \qquad (2.12a)$$

$$= \frac{1-(1-\Omega(y))^t}{\Omega(y)} \alpha_{max}(y,t) \qquad (2.12b)$$

So, for additive selection, $\Omega(y) = (S+R)y$ and:

$$U(y,t) = 2R \, \frac{1-(1-(S+R)y)^t}{(S+R)y} \left( \int_y^1 \Psi_{eq}(z)dz + M \right) \qquad (2.13)$$

$\Psi_{eq}(y)$ under additive selection is shown in Eq. 2.6. We can also numerically solve for $\Psi_{eq}(y)$ under an arbitrary selection function, and finding $\Omega(y)$ for different forms of selection is trivial, hence we can calculate U(y,t) for any selection function. Now considering first that $\alpha_{max}(y,t)$ is derived from the maximum number of blocks larger than y at any time, $\Psi_{eq}(y)$, and second that we now have an upper bound for the number of blocks at a particular time t which is the same as or lower than $\Psi_{eq}(y)$, then we can calculate a new lower value for $\alpha_{max}$ as:

$$\alpha'_{max}(y,t) = 2R \int_y^1 U(z,t)dz + 2RM \qquad (2.14)$$

Hence a new upper bound U'(y,t) is found, which will be closer than U(y,t) to the true distribution at time t. Eq.s 2.11 and 2.12b

can be re-applied to each other in this way, until the upper bound converges on its closest approach to the true distribution. Fig. 2.3 compares successive calculations of the upper bound with Barton's solution for additive selection, and simulation results. The difference between the upper bound best estimate and Barton's solution is constant over time, and seems to depend entirely on how rapidly block numbers rise with decreasing size for the equilibrium distribution from which the upper bound is initially derived: the slower this rise the better the estimate. This makes intuitive sense as at the limit where there is no change in block numbers with size, the real distribution for time t, the upper bound, and the equilibrium distribution would all coincide. We would expect the upper bound estimate to be most accurate then for strong coupling, which produces a shallow increase in block numbers with decreasing block size at equilibrium.

The stochastic nature of the evolution of the distribution of small blocks is apparent in Fig. 2.3. The probability of a block size y being produced by recombination of a chromosome containing a block size $x > y$ is simply Ry. Thus even for high recombination rates, R>0.5, we would expect blocks of size $\approx e^{-10}$ to be produced less than once every $10^5$ reproduction events. As the total number of hybrid individuals after 2500 generations of simulation is of order $10^4$ (for $\theta = 4$), such blocks will only be produced rarely, and though under very little selection ($S \approx 5 \times 10^{-5}$), will be subject to drift, either quickly disappearing, or increasing. The predicted density distribution functions of Fig. 4 are therefore, in the case of small blocks, realised in the

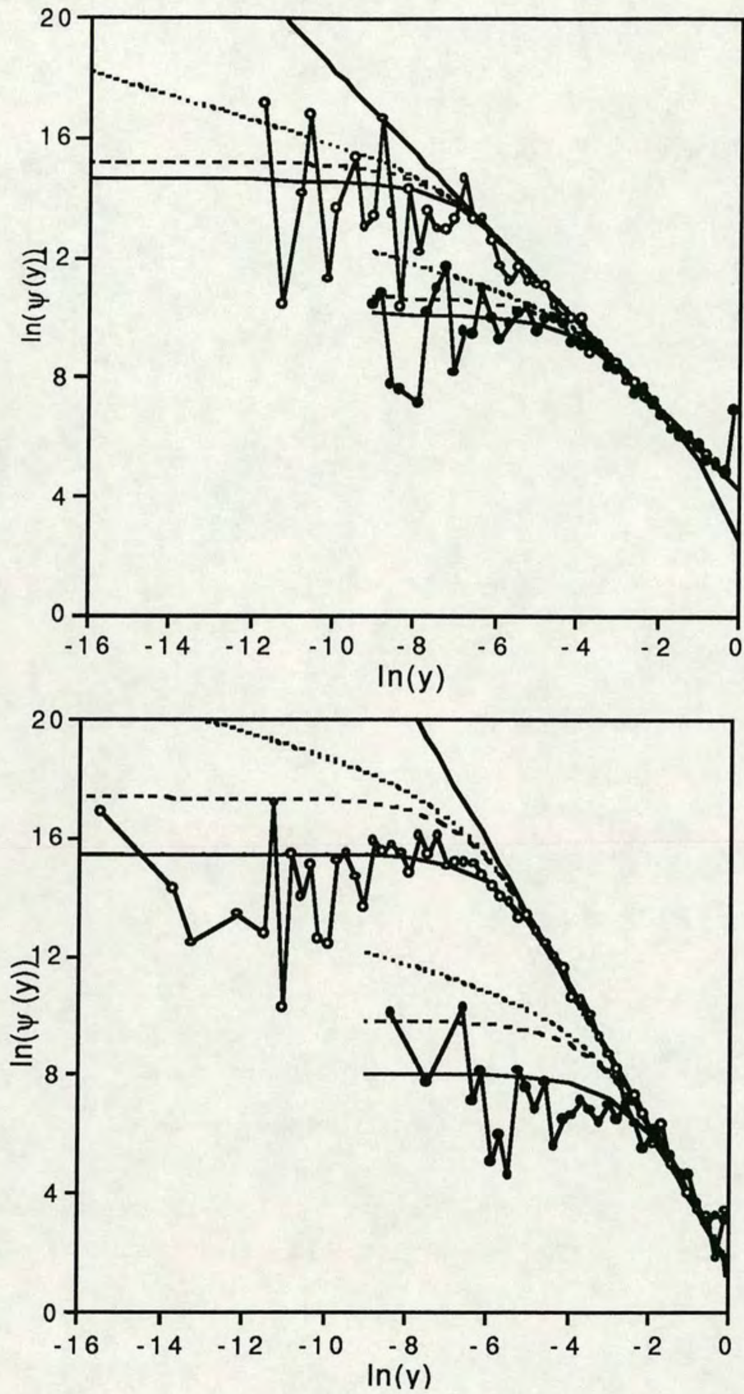



Figure 2.3: The distributions of figure 2.2 with Barton's solution for additive fitness (solid line), and upper bounds (dotted lines) overlaid for both t=2500 and t=100 generations. For $\theta = 4$ (above) U(y,t) and U'(y,t) are shown. U"(y,t) coincides with U'(y,t) at this resolution. For $\theta = 0.5$ (below) U"(y,t) and U'''(y,t) are shown. U'''(y,t) coincides with U"(y,t) at this resolution.

simulations by a very few blocks subject mainly to drift. The way in which blocks are censused in the simulation also means as block size decreases there are an increasing number of size classes to which no blocks are assigned. (calculated densities for classes which contained no blocks are not shown, for clarity).

Both the upper bound and Barton's solution for additive selection can be scaled by R, and include further terms in R. This dependence on R contrasts with the situation at equilibrium where block numbers depend solely on the coupling, $\theta = S/R$. This implies that two populations with the same coupling ratio, but different values of S and R will tend to the same equilibrium, but at different rates. As we might expect increasing R speeds the approach of populations to equilibrium, but successive increases in R have less effect on the ln/ln scale, so that even under high rates of recombination the progress toward equilibrium remains of the same order.

## 2.5 Discussion

Analyses in population genetics have for the most part considered the limit of either a few discrete loci ( e.g. Christiansen and Feldman 1975, Karlin 1975) or a very large number of loci, the relative simplicity of these extremes favouring exact solutions. Where problems with a few discrete loci become complex, explicit computer simulation of gametes is often useful. The equivalent simulations of the continuous case have rarely been carried out, yet the same possibilities for

testing and extending the analytic work explored in the present study exist in many other areas of population genetics.

The simulations of infinite locus clines indicate that their approach to equilibrium is very slow: it takes a very long time for recombination to thoroughly mix the haplotypes of two such divergent populations. This point is illustrated in Fig. 2.4, which shows the average pairwise linkage disequilibrium over 20 evenly spaced loci during the first 1000 generations of two sets of simulations identical to those used previously. The average pairwise disequilibrium was calculated from the variance of the hybrid index, and the variance of allele frequency across the 20 loci for those individuals followed in the simulation, following Eq. (2b) of Barton and Gale (1993). The decay of linkage disequilibrium is initially rapid, but becomes very slow, in contrast with the evolution of the distribution of blocks, which progresses at a steady rate.

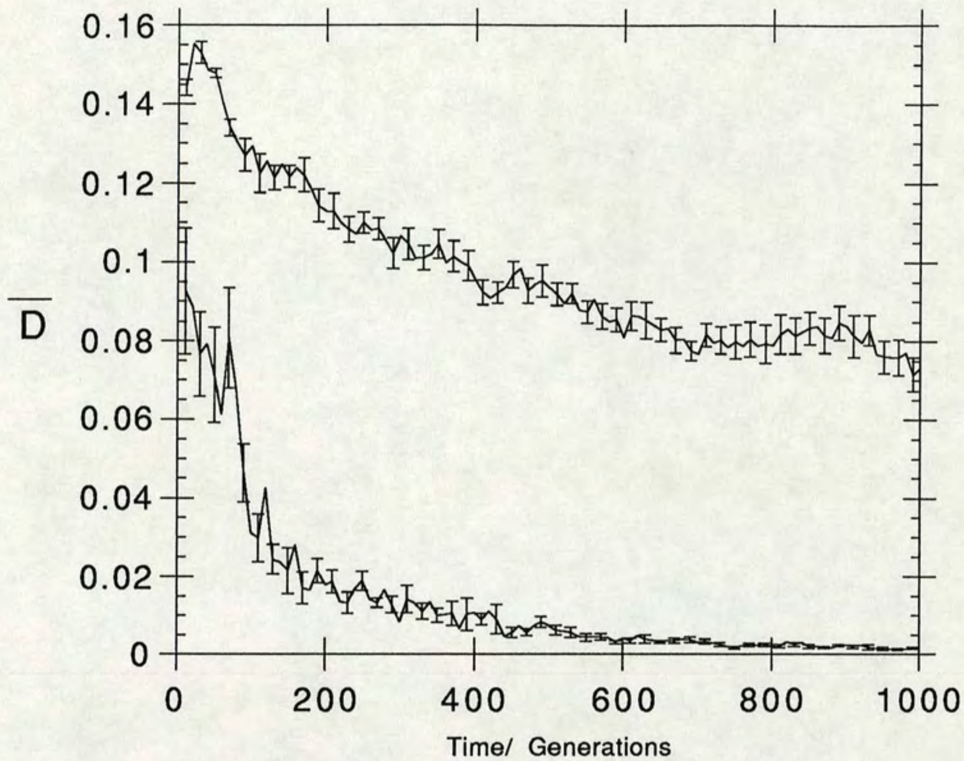



Figure 2.4: The average pairwise linkage disequilibrium ($\bar{D}$) between twenty marker loci spaced uniformly along the length of the chromosome, measured over all the hybrid individuals in the population during the first 1000 generations of simulation. [Upper line] Strong coupling ($\theta = 4$). [Lower line] Weak coupling ($\theta=0.5$). Results are averaged over 10 replicates. Standard errors are shown for every second observation.

The steady evolution of the distribution of blocks allows us to census blocks in a simulation population under additive selection and by comparison with Barton's solution for the distribution at time t, estimate the number of generations since secondary contact for that population. More generally we can census blocks from a simulation population under an arbitrary unknown selection regime, and provided the selection function is smooth, by comparison with the upper bound on the distribution of blocks we can set a lower bound on the number of generations since secondary contact. This ability suggests it may be possible to develop a method from this analysis for dating natural hybrid zones. This is the topic explored in the next chapter.

## 2.6 Summary

Fisher's method of junctions is used to investigate the degree of association between selected alleles in a cline, in the limit where there is divergence between very many genes. A computer model is used to simulate one of a pair of infinite demes which exchange individuals each generation. Selection is on haploids, is additive and is equivalent to heterozygote disadvantage. Recombination is uniform over a single chromosome. A 'critical value' of selection exists at equilibrium, below which loci act independently and above which they act in association (Barton, 1983). Starting with secondary contact, simulation results contrast markedly with the equilibrium solution. The 'critical value' is not apparent in the simulated clines, even after many generations. Rather, loci

remain associated to some extent under all degrees of selection. The simulation is consistent with the equilibrium analysis in all other respects, and therefore indicates that under weak selection the approach to equilibrium is very slow. This is borne out by further numerical calculations. The slow approach to equilibrium enables us to estimate the time since contact between two demes under idealised conditions.

# Appendix 2.1: The Fate of Introgressing Chromosomes

THIS APPENDIX WAS COMPOSED BY N. H. BARTON

In a departure from the notation of Barton (1983), L is used here to denote number of loci instead of n. We will first derive the solution for L discrete loci, and then take the limit of large L to find results comparable with the simulations. When a block of L genes enters a population at low frequency, it will be broken up by recombination; the fragments will then be eliminated by selection at a rate proportional to their size (Eq. 1 of Barton, 1983). After time t, the parent block will have produced a distribution of daughters, $f(i,t)$ $(0<i<L)$. (This distribution is scaled so that if the frequency is initially $p(L) = A$, it will be $p(i,t) = Af(i,t)$ after time t). This distribution can be used to solve a variety of problems. For example, if blocks of size L have entered at a steady rate m for t generations, the distribution can be found by integrating over the products of introgression t generations back:

$$p(i) = m \int_0^t f(i,\tau)\, d\tau \qquad \text{(A2.1)}$$

A block of i genes is eliminated by selection at a rate si; because it spans (i-1)r map units, it is broken up by recombination at a rate r(i-1). Blocks of size i are generated by recombination from larger blocks:

$$\frac{df(i,t)}{dt} = -[si + r(i-1)]f(i,t) + 2r \sum_{j=i+1}^{L} f(j,t) \qquad \text{(A2.2)}$$

This is Eq. 1 of Barton, 1983, and is the discrete analogue of Eq. 1 above. The initial conditions are that $f(i,0) = 0$ for $i<L$, and $f(L,0) = 1$. Blocks of length i are eliminated at rate $\lambda_i^2 = [si + r(i-1)]$, and so the solution is a sum of components, each decaying exponentially at a characteristic rate $\lambda_i^2$. (Formally, the sum over the set of eigenvectors of Eq. A2.2, each decaying at a rate given by the eigenvalues $\lambda_i^2$):

$$f(i,t) = \sum_{j=i}^{L} b(i,j)e^{-\lambda_j^2 t} \qquad \text{(A2.3a)}$$

where $b(i,j) = \binom{j-i-2\gamma-1}{j-i}\binom{L-j+2\gamma-1}{L-j}$,

$$\gamma = \frac{1}{(1+\theta)}, \quad \lambda_j^2 = (sj + r(j-1)) \qquad \text{(A2.3b)}$$

The binomial coefficient, $\binom{a}{b}$, is defined for arbitrary a,b in terms of Gamma functions:

$$\binom{a}{b} = \frac{\Gamma(a+1)}{\Gamma(b+1)\Gamma(a-b+1)} \qquad \text{(A2.3c)}$$

This solution was checked by using the symbolic computation package Mathematica (Wolfram, 1992) to generate f(i,t) for various values of L, and inserting this expression back into Eq. A2.2.

We now seek the limit $L \rightarrow \infty$, keeping $x=i/L$ of order 1. First, consider the contribution of terms with $y=j/L$ and $(y-x)$ of order 1. Let $T = Rt = rLt$, $F(x,T) = f(xL,T/R)$, and $B(x,y) = n^2$ $b(xL,yL)$. Taking the limit of $b(i,j)$ gives:

$$B(x, y) = \frac{(1-y)^{2\gamma-1}(y-x)^{-2\gamma-1}}{\Gamma(2\gamma)\Gamma(-2\gamma)} \qquad \text{(A2.4)}$$

$F(x, T)$ should now be given by the continuous limit of Eq. A2.2a:

$$F(x, T) = \backslash\mathrm{i}(x,1,\ e^{-y(1+\theta)T}\ B(x,y)\ dy \qquad \text{(A2.5)}$$

However, this integral diverges near $y = x$. This problem can be avoided by by splicing together the continuous integral from $x+\varepsilon$ to 1, and the discrete sum from $j = i$ to $i+L\varepsilon$:

$$F(x, T) = \int_{x+\varepsilon}^{1} e^{-y(1+\theta)T} B(x,y)\, dy \quad + \qquad \text{(A2.6)}$$

$$e^{-x(1+\theta)T} \lim_{L\rightarrow\infty}\left( L \sum_{k=0}^{L\varepsilon} \binom{k-2\gamma-1}{k}\binom{L-i-k+2\gamma-1}{L-i-k} e^{-k(1+\theta)T/L} \right)$$

Since $1<<L\varepsilon<<L$, and $k<L\varepsilon<<L$, the sum can be approximated by dropping the factor $\exp(-k(1+\theta)T/L)$, and by approximating $\binom{L-i-k+2\gamma-1}{L-i-k}$ using Stirling's approximation, by $(L-i)^{2\gamma-1}/\Gamma(2\gamma)$:

$$F(x, T) = \int_{x+\varepsilon}^{1} e^{-y(1+\theta)T} \frac{(1-y)^{2\gamma-1}(y-x)^{-2\gamma-1}}{\Gamma(2\gamma)\Gamma(-2\gamma)} dy \quad + \tag{A2.7}$$

$$e^{-x(1+\theta)T} \lim_{L\to\infty}\left( L \frac{(L-i)^{2\gamma-1}}{\Gamma(2\gamma)} \sum_{k=0}^{L\varepsilon} \binom{k-2\gamma-1}{k} \right)$$

The sum has an explicit solution:

$$\sum_{k=0}^{L\varepsilon} \binom{k-2\gamma-1}{k} = \binom{L\varepsilon-2\gamma}{L\varepsilon} \tag{A2.8a}$$

For large $L\varepsilon$, this converges to:

$$= \frac{(L\varepsilon)^{-2\gamma}}{\Gamma(1-2\gamma)} \quad \text{for } L\varepsilon >> 1 \tag{A2.8b}$$

The integral can be simplified by dividing it into convergent and divergent components:

$$F(x, T) = \tag{A2.9}$$

$$\int_{x+\varepsilon}^{1} \frac{\left(e^{-y(1+\theta)T}(1-y)^{2\gamma-1} - e^{-x(1+\theta)T}(1-x)^{2\gamma-1}\right)(y-x)^{-2\gamma-1}}{\Gamma(2\gamma)\Gamma(-2\gamma)} dy$$

$$+ \; e^{-x(1+\theta)T}(1-x)^{2\gamma-1} \int_{x+\varepsilon}^{1} \frac{(y-x)^{-2\gamma-1}}{\Gamma(2\gamma)\Gamma(-2\gamma)} dy$$

$$+\ e^{-x(1+\theta)T} \lim_{L\to\infty}\left( L\ \frac{(L-i)^{2\gamma-1}}{\Gamma(2\gamma)}\ \frac{(L\varepsilon)^{-2\gamma}}{\Gamma(1-2\gamma)} \right)$$

$$= \int_x^1 \frac{\left(e^{-y(1+\theta)T}(1-y)^{2\gamma-1} - e^{-x(1+\theta)T}(1-x)^{2\gamma-1}\right)(y-x)^{-2\gamma-1}}{\Gamma(2\gamma)\Gamma(-2\gamma)}\,dy$$

$$+\ e^{-x(1+\theta)T}(1-x)^{2\gamma-1}\left[\frac{(1-x)^{-2\gamma}-\varepsilon^{-2\gamma}}{(-2\gamma)\Gamma(2\gamma)\Gamma(-2\gamma)}\right]$$

$$+\ e^{-x(1+\theta)T}(1-x)^{2\gamma-1}\,\frac{\varepsilon^{-2\gamma}}{(-2\gamma)\Gamma(2\gamma)\Gamma(-2\gamma)}$$

(A2.10)

The singular components, which involve $\varepsilon$, cancel as expected, and the remaining integral can be evaluated explicitly. Using the identities $\Gamma(1-w)\Gamma(w)\sin(\pi w) = \pi$, and $(-2\gamma)\Gamma(-2\gamma) = \Gamma(1-2\gamma)$, the distribution of block sizes in the continuous limit is:

$$F(x, T) = T\ e^{-x(1+\theta)T}\ {}_1F_1[(\theta-1)/(\theta+1), 2, -T(1+\theta)(1-x)]$$

(A2.11)

This solution was verified by substitution into Eq. A2.1. It is a good approximation even for moderately small L ($\approx 10$, say).

The distribution of block sizes can be integrated to give the total frequency of chromosomes carrying some introgressed material, and the total frequency of introgressed genes. A contribution

must also be added from the intact chromosomes (x=1), which are at frequency $\exp(-T(1+\theta))$:

$$\int_0^1 F(x\ T)\,dx = {}_1F_1[(\theta-1)/(\theta+1),1,-T(1+\theta)] \qquad \text{(A2.12a)}$$

$$\int_0^1 x\ F(x\ T)\,dx = \lim_{\varepsilon\to\infty}\Big(\varepsilon\ {}_1F_1[(\theta-1)/(\theta+1),\varepsilon,-T(1+\theta)]\Big) \qquad \text{(A2.12b)}$$

Where ${}_1F_1$ is the hypergeometric function.

# (3)
# Estimating the Age of Hybrid Zones

"..like proposing a method for measuring the rotational veloc
saucers."

(M. Slatkin, pers. comm.)

3.0 Introduction ....................................................................................45
3.2 Additive Selection...............................................................................48
3.2 Arbitrary Selection...............................................................................50
3.3 Modelling Finite Populations - a Preliminary Study ..................54
3.3.1 The Junctions Model.........................................................56
3.3.2 Results ..............................................................................58
3.3 Discussion ...........................................................................................61

# 3.0 Introduction

The analyses of multilocus clines developed in the previous chapter could give us new insights about hybrid zones: Given a map length and selection function for a simulated population we can at least place an upper bound on, and for the case of additive selection we have an exact solution for, the distribution of the sizes of homogeneous blocks at any time. This chapter considers the applications of this method. From an observed distribution of block sizes within a known map length, what could we infer about selection and the number of generations since secondary contact? Could we distinguish between secondary contact and divergence in parapatry? It may seem premature to ask such questions before extending the model to consider finite populations, exchange over a continuum, and non-uniform selective effects on the chromosome, or in fact whether it is feasible to measure the distribution of blocks in a natural population at all. Such misgivings prompted the quote at the head of this chapter, and it is true that the population for which the analysis is described is far removed from reality. However, extending the model will involve much work and much simulation, so it seems sensible first to show that the method works *in principle* for flying saucers, before we generalise to frisbees. It is encouraging that the results presented here seem robust to the kind of selection against hybrids.

The recursive calculation of the upper bound on the distribution of blocks appears an extremely powerful tool: it is intuitively simple, straightforward to calculate, and can be applied for *any fitness function.* At present it is unclear why the upper bound for additive selection does not converge exactly on Barton's solution for additive selection. The difference between the upper bound best estimate and Barton's solution is constant over time (figure 2.3), and constant for a given coupling coefficient (Figure 3.1). The equilibrium distribution of blocks from which the upper bound is initially derived is the only part of the derivation which depends only on the coupling coefficient. It is possible therefore that the discrepancy between the two solutions is a function of the recursive derivation of the 'flat' portion of the distribution for smaller blocks from the 'linear slope' at equilibrium: slow convergence of the upper bound for larger blocks will lead to extremely slow convergence for smaller blocks. If this is correct, it places a practical limit on the accuracy of the upper bound. We shall assume this is the case in the following work. If so it seems possible that a thorough study of the properties of the recursion might reveal a simple method of correcting for this error.

Using the upper bound calculation as it stands, we can develop ways to estimate the age of a cline under additive selection from the observed distribution of blocks: The gradient of linear slope of the distribution allows us to estimate the coupling $\theta$, and its intercept allows us to estimate dispersal M. Given these estimates we can match our expectation for the distribution of blocks at different times to the observed distribution. If the cline is subject

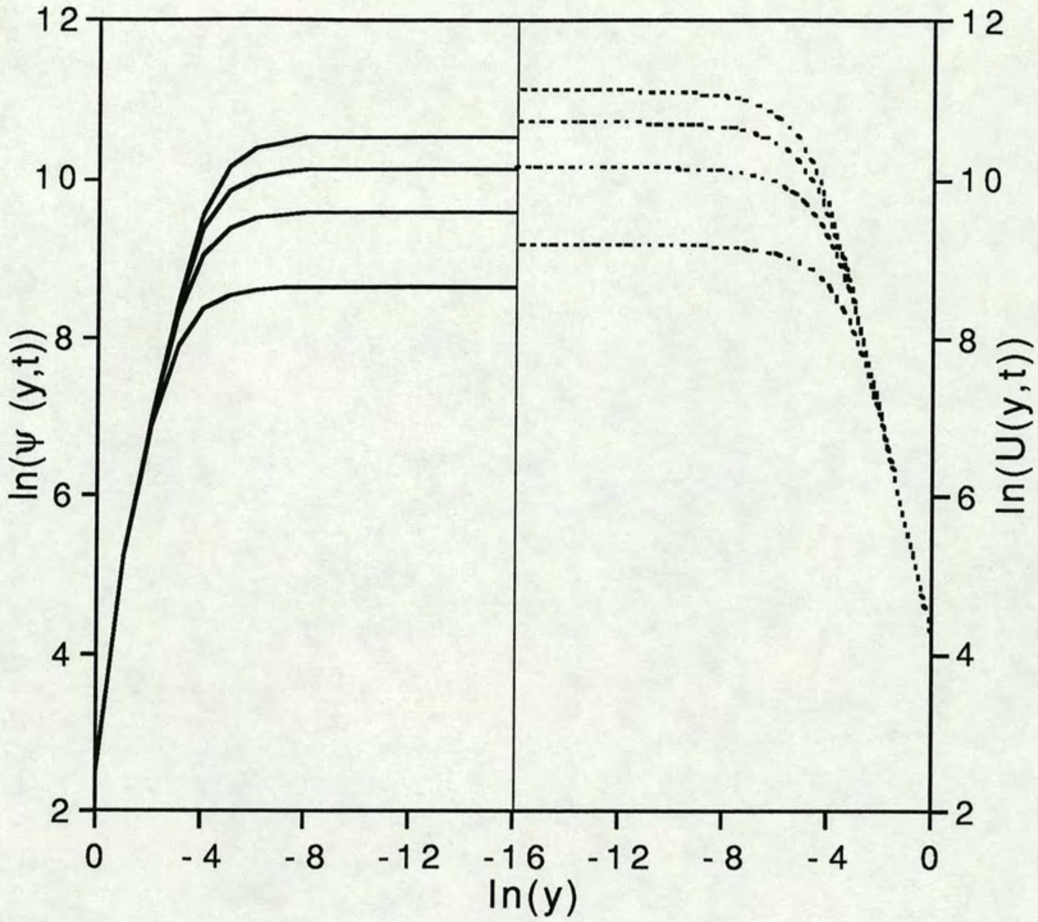



Figure 3.1: Barton's solution for additive fitness (solid lines), and the upper bound best estimate (dotted lines), after 100 generations with coupling $\theta$ = 4. From top to bottom the plots are for R=0.20, 0.15, 0.10 and 0.05. The x axis is reflected for ease of comparison of corresponding plots.

to some unknown form of selection against hybrids, it still may be feasible to apply this method. Whatever form selection takes, very small blocks are likely to be generated by a linear portion of the fitness function. We shall develop these ideas by first considering the details for additive selection, and then extending our argument to arbitrary selection functions.

## 3.2 Additive Selection

Following the same notation as in the previous chapter, let us first assume an observed distribution of block sizes $\varepsilon(y)$ has been evolving under steady conditions of additive selection Sy for $\tau$ generations, and that we are observing a map length R. Unless $\tau$ is very small, there should be a linear portion of the distribution, on a ln/ln scale, where blocks have reached their equilibrium numbers. Eq. 2.6 for $\Psi_{eq}(y)$ tells us that the gradient of this linear portion is $(3+\theta)/(1+\theta)$, and its y-intercept is $2M\theta/S(1+\theta)^2$. If we know the length of the map (R) and we estimate $\theta=S/R$ and M from the slope and intercept, we can estimate the number of generations since secondary contact by calculating Barton's distribution of block sizes under additive fitness as a function of R, S and M for increasing numbers of generations t until the solution best matches the observed distribution.

This approach is straightforward, but unlikely to be of any practical use, as it is assumed that we know the form of selection *a priori*. To relax this limitation we can instead use the upper bound which applies for arbitrary selection. It is convenient

however, to first develop the method for the additive selection assumptions above, and then generalise it. As before S and M can be calculated from the gradient and y-intercept of the linear portion of $\varepsilon(y)$. We can then formulate an upper bound U(y,t) from $\alpha_{max}$ as in Eq. 2.10, with $\Omega(y)=Sy+Ry$. If we now calculate U(y,t) for an increasing number of generations t, until it most closely approximates $\varepsilon(y)$, then t will be a lower bound on the number of generations since contact. The advantage of initially considering additive selection is that we can assess the accuracy of this method by manufacturing observed distributions $\varepsilon(y)$ using Barton's solution for the distribution at a given time, and comparing this to the number of generations estimated using the upper bound. We have seen (Figure 2.3) that the accuracy of U(y,t) is influenced by $\theta = S/R$. Figure 3.2 summarises the effect of $\theta$ on the upper bound estimate after 100 and 2500 generations. As expected the estimates are closest to the true number of generations when coupling is strong, and the percentage error remains roughly constant over time.

In Chapter 2 we noted that the distributions produced by the simulations are subject to stochastic fluctuations for small blocks. If we are to use the above method for real data then we must be able to compensate for such noise. If we wish to find the value of $\Psi(y,t)$ for the plateau of small blocks, then we might take the average block count over a range of block sizes we believe to be at the plateau. However, the stochastic fluctuations are by their nature far from normally distributed around the mean, making it a poor approximation of the true value of $\Psi(y,t)$. Analysing the

distribution of fluctuations appears complicated, so as a first guess the following method was used for estimating $\Psi(y,t)$ from the simulation data. The point on the distribution where block numbers no longer auto-correlate as size decreases was taken as the start of the plateau. The mean and standard deviation of the non-zero block counts below this point were calculated. Any points more than one standard deviation from the mean were then discarded from the analysis. $\Psi(y,t)$ was estimated as the mean of the remaining points. The resulting lower bounds calculated from the simulated distributions after 100 and 2500 generations for $\theta=4$ are 81 and 1303. The same bounds for $\theta=0.5$ are 38 and 1101. These estimates are shown in Fig. 3.2 as percentages of the actual number of generations. It seems clear that we can compensate for the fluctuations over small block sizes when calculating lower bounds on t, though they must be taken into account as a source of error.

## 3.2 Arbitrary Selection

To extend this method for arbitrary selection we note that the characteristics of equilibrium block distributions for many selection functions are similar to that for additive selection: the distributions are dominated by many very small blocks, and as we consider the interval of sizes between very small and zero, any smooth selection function will tend to linearity. The effect of an arbitrary selection function S(y) on small blocks will then approximate to additive selection equal to the gradient of S(y) as

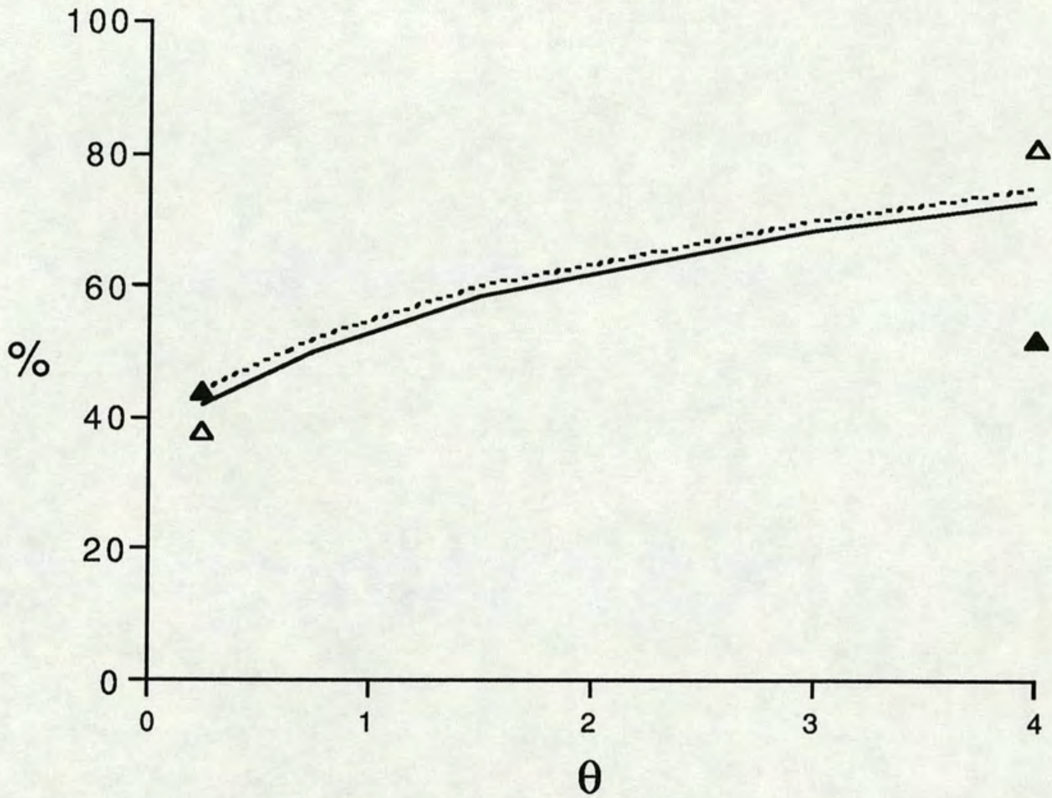



Figure 3.2: The lower bound $\tau$ for number of generations since contact expressed as a percentage of t, the actual number of generations. Lines: lower bounds calculated by fitting U""'(y,$\tau$) to Barton's solution for the distribution at time t. Dotted line: t=100; Solid line: t=2500. Triangles: lower bounds calculated by fitting U""'(y,$\tau$) to $\Psi$(y,t) estimated from the simulation distributions shown in Fig. 2.3. Open triangles: t=100; Solid triangles: t=2500.

it tends to zero. Fig. 3.3 compares numerically calculated equilibrium distributions for a number of selection functions to the equivalent distributions for additive selection, and demonstrates the remarkably close fit for small blocks.

Let us now assume that some distribution of block sizes $\varepsilon(y)$ has been evolving under steady conditions of some unknown selection $S(y)$ for $\tau$ generations, and again we are observing a map length R. No matter what the selection function, as long as it is smooth there should be a linear portion of the distribution for small blocks, on a ln/ln scale, where blocks have reached equilibrium, just as for additive selection. As before, by comparison with Eq. 6 for $\Psi_{eq}(y)$, the gradient of the linear portion allows us to estimate the coupling $\theta$ for this portion of the distribution, and hence $S(y)$ for small blocks. We know nothing about the selection acting on larger blocks outside the linear portion of the distribution, but Eq. 2.11 for the upper bound on block numbers refers only to selection on blocks of size y, in $\Omega(y)$. No assumptions about the selection function above its linear portion are necessary; $\alpha_{max}$ can be calculated as in equation 10, so $U(y,t)$ can be calculated for small blocks in the same manner as for additive selection. Hence we can find a lower bound on the number of generations since contact from an observed distribution of block sizes with no knowledge of the selection acting. We need only assume the selection function is smooth.

To try out this method for arbitrary selection, a series of simulations were carried out using one of the complex fitness

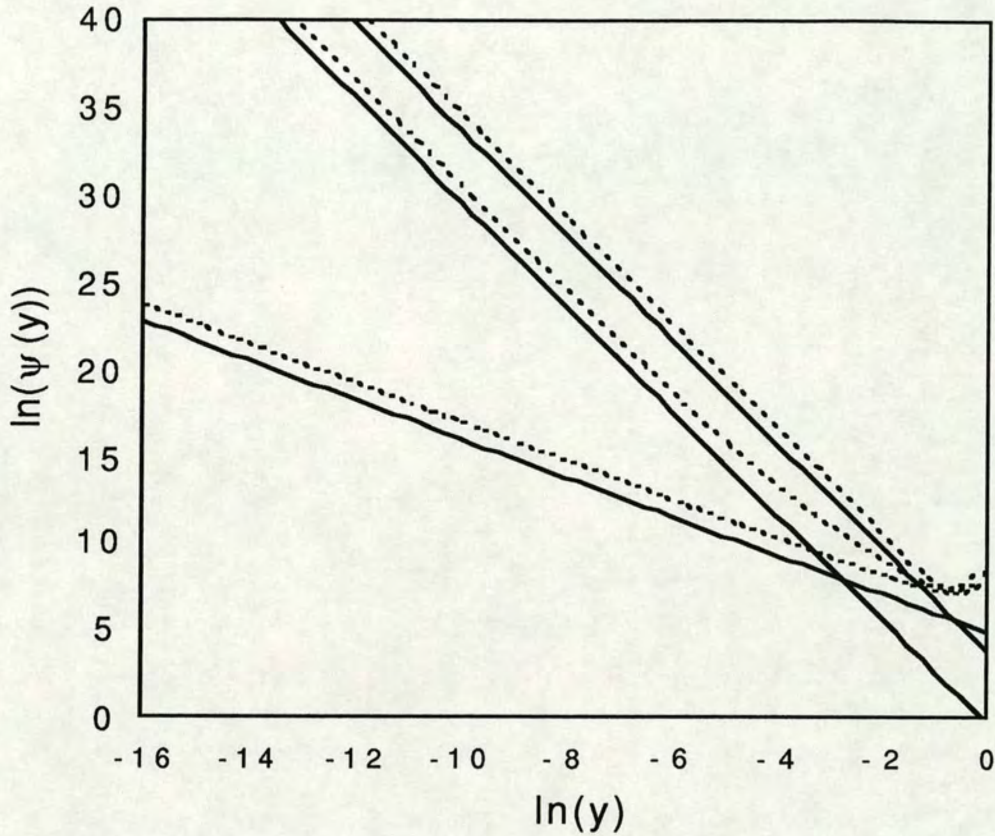



Figure 3.3: Comparison of the equilibrium distributions of blocks with complex fitness functions to distributions under the additive fitness function. Dotted lines: data for fitness functions of the form $s(y)= S(4y(1-y))^{\beta}$, where S=4. Bottom: $\beta=1$ (quadratic function), middle: $\beta=2$, top: $\beta=4$. Solid lines: Distributions for additive fitness, where S is equal to the gradient of $s(y)=S(4y(1-y))^{\beta}$ as y tends to zero, where S=4. Bottom: $\beta=1$ (quadratic function), middle: $\beta=2$, top: $\beta=4$. The y-intercepts of these distributions have been adjusted by altering M, so they lie alongside the distributions for complex fitness functions, for ease of comparison.

functions from Fig. 3.4, which perhaps more closely approximates the situation in a real hybrid zone: pure genomes from either population are equally fit, F1's are least likely to be fit, and back-crossing of small blocks onto either background will produce progressively fitter individuals. The fitness reduction due to small blocks tends to zero with decreasing block size, so that the equivalent additive selection function for small blocks is S=0. Thus $\theta = 0$, and we would expect the difference between the upper bound and the true distribution to be at its maximum. The simulations were run for 2500 generations producing the distribution of block sizes shown in Fig. 3.4. On comparison with an upper bound calculated for additive selection with S=0, a lower bound for generations since contact was estimated as t=853, or 34.1% of the true age. This degree of error seems consistent with what we might expect for $\theta = 0$, by comparison with the earlier analysis of distributions created under additive selection (Fig. 2.6).

## 3.3 Modelling Finite Populations - a Preliminary Study

The consequences of modelling a finite population are manifold. In the last chapter we modelled blocks of immigrant genetic material which were rare in an infinite native population. In this case immigrant alleles will almost always be heterozygous, and so additive heterozygous disadvantage, with fitness (1-Ys), where Y is the number of heterozygous loci is equivalent to a haplotype fitness (1-Ys). Thus selection on haplotypes can be used for simplicity (see section 2.1). However, if individuals disperse into

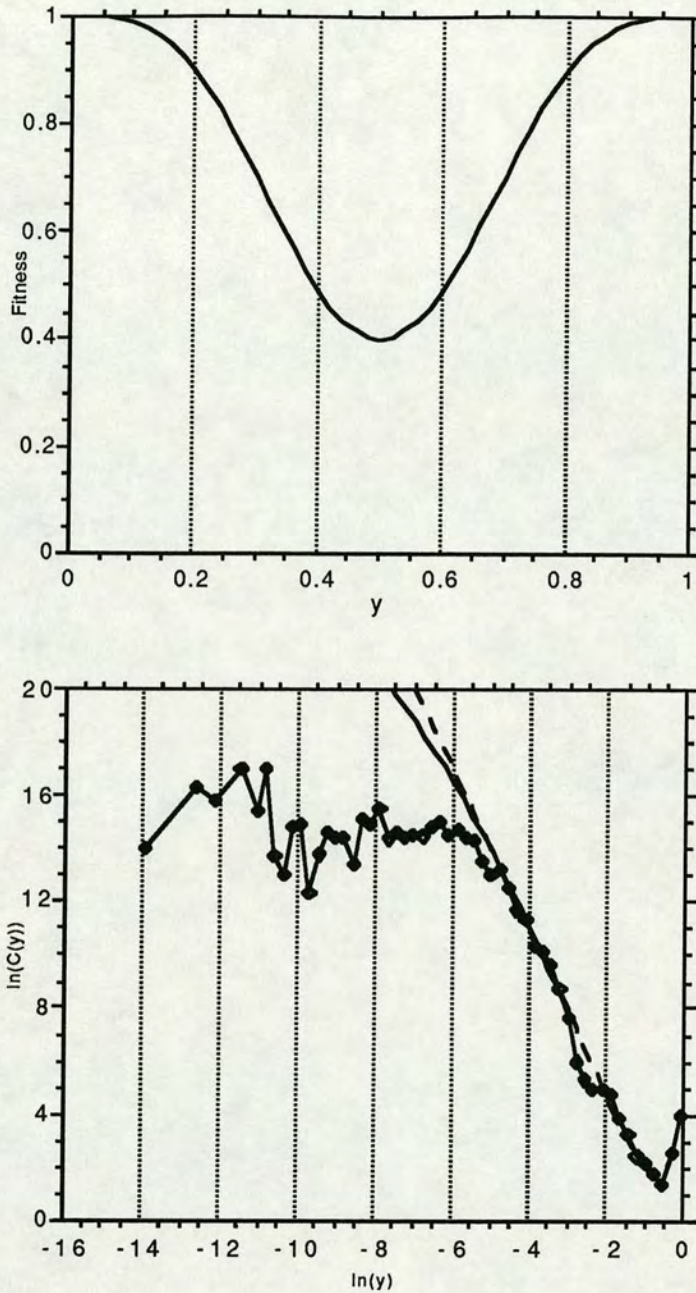



Figure 3.4: Above: Fitness against ratio of immigrant genome for the selection function $s(y)= S(4y(1-y))^{\beta}$ where S=0.6, and $\beta$ =4. Below: Data from simulations using the fitness function above. Diamonds: The observed distribution of block sizes found after 2500 generations. Dotted line: The numerical equilibrium solution for the distribution of block sizes. Solid line: The upper bound for the number of blocks after 2500 generations (from Eq. 2.8). Simulation results are averages over ten replicates.

a *finite* population, blocks of immigrant genetic material may become common and interact. We can no longer assume blocks are heterozygous, and so we must develop a diploid model. The average fitness $\bar{w}$ of the infinite populations previously considered was assumed to be unaffected by the rare blocks entering the population. As blocks can become common in a finite population we must calculate individual fitnesses relative to $\bar{w}$. The changes necessary in the implementation of simulations are incorporated in Appendix 2.

### 3.3.1 The Junctions Model

As the first stage in extending the model to exchange across a chain of finite demes, this preliminary study considers a finite deme of 500 diploid individuals, initially all haplotype A, which is flanked by two infinite demes, one of which is all haplotype A, the other all haplotype B. As before, generations are discrete with migration followed by selection and then recombination. Censuses are taken after recombination. In each generation the three demes exchange M individuals, chosen at random, with each of their nearest neighbours. Immigrants from either of the flanking demes are always assumed to pure natives of those demes, and only the central deme is followed explicitly.

Selection is by heterozygote disadvantage. The proportion of the genome which is heterozygous (h) is calculated for each individual by comparison of junction positions along the diploid

genome. Fitnesses are calculated using an equation of the form $S(4h(1-h))^{\beta}$, equivalent to the fitness function $S(4y(1-y))^{\beta}$ used in section 3.2 (see figure 3.4) when applied to blocks size y which are always heterozygous. Relative fitnesses are implemented after Gale in (Barton & Gale, 1993). A table $C_1..C_N$ of the cumulated $C_i$ of the fitness of each member i of the population is created for each generation. A parent of the next generation is chosen by pulling a random number uniform in $(0,C_N)$; the interval of the table in which the random number lies indicates the chosen individual. Thus the probability of parenthood is precisely the relative fitness, and parents are chosen by sampling with replacement. 2N parents are drawn from the population, N pairings each produce one offspring to form the next generation of N individuals.

Recombination is uniform along the chromosome, with a number of chiasmata drawn from a Poisson distribution with expectation R, where R is the total map length and assuming no interference (See Appendix 1,1.0 for discussion). The positions of all chiasmata for a chromosome pairing are drawn, ordered and then applied to the lists of junctions of the parents, to produce a single offspring which is added to the individuals of the next generation. The same tests were applied to the model as to the model in the previous chapter. In addition, as the finite deme receives equal dispersal from its two neighbours we would expect the hybrid index (defined as the proportion of the genome which is of the same state as the initial population of the deme) to ultimately

tend to 1/2. The model satisfies this expectation (Figure 3.5): the hybrid index initially drops rapidly, then more slowly.

### 3.3.2 Results

Figure 3.6 shows the distribution of block sizes after 1000 generations for a simulated population receiving M=10 individuals from flanking demes, R=0.15, S=0.0375, $\beta$=1. This contrasts with the distribution found in an infinite population under similar selective conditions (Figure 3.4): The distribution is similar to that of an infinite population with a much higher coupling coefficient. If the interaction of blocks in finite populations serves to reduce the gradient of the equilibrium distribution of blocks then we would expect an upper bound approach to give more accurate estimates of age for finite populations, all other things being equal.

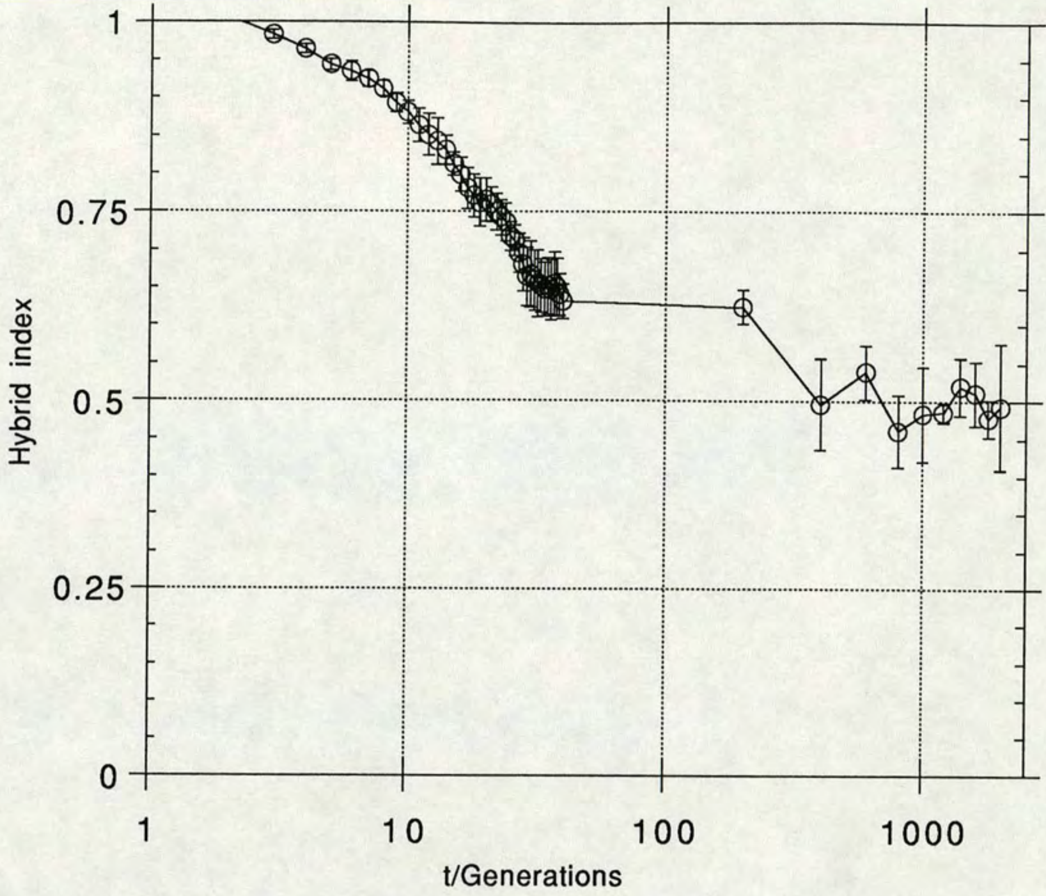



Figure 3.5: The change in hybrid index over time (plotted on a log scale) for a simulated deme, M=10, R=0.15, S=0.0375, β=1. Results are averages over 6 replicates, error bars indicate standard deviations.

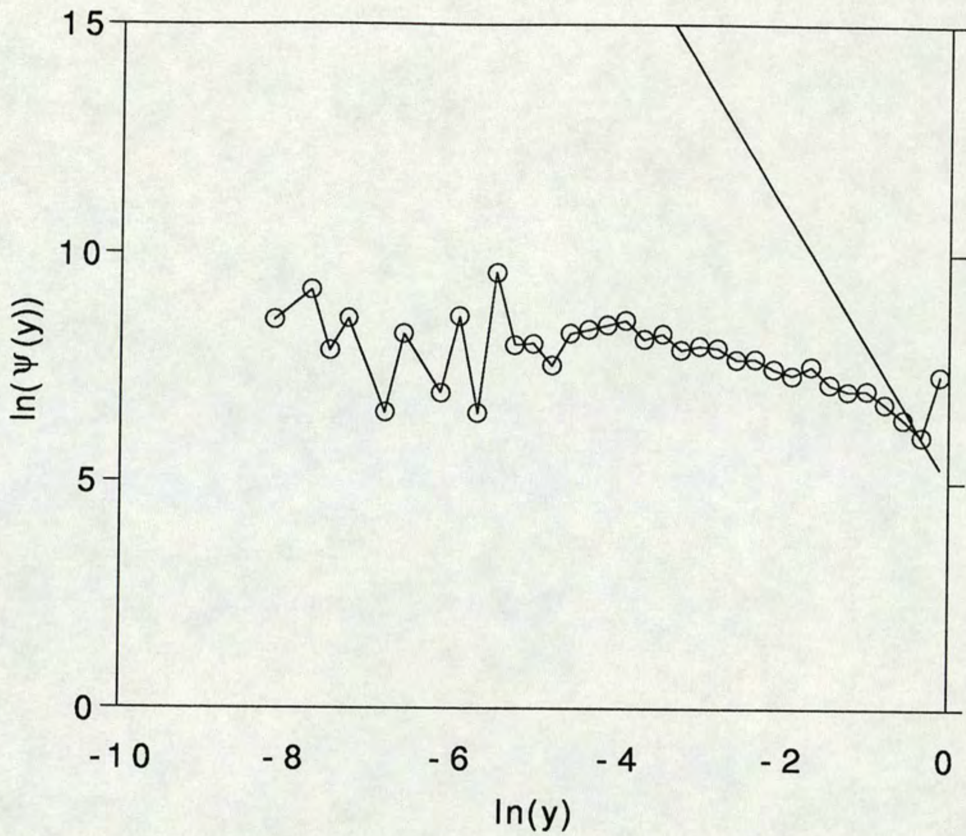



Figure 3.6: Comparison of observed block distributions to equilibrium prediction. Circles: The observed distribution of block sizes found after 1000 generations. Bold line: The equilibrium solution for the distribution of block sizes (from Eq. 2.5). Simulation results are averages over six replicates.

## 3.3 Discussion

These preliminary studies suggest that we may be able to develop a method for dating natural hybrid zones. It appears possible in principle to estimate time since secondary contact from an observed distribution of block sizes. A useful method would have to consider the implications of finite population size, exchange across a continuum and non-uniform selective effects across the genome. The initial work on extending the simulation model to finite populations is not discouraging, but also highlights the extent of work necessary in developing a full treatment of the problem.

Progress in sequencing methods mean that it may be possible to collect information on the distribution of block sizes in a region of known map length from individuals in a real hybrid zone. Work of this nature has already been done on prokaryotes (Maynard Smith 1990 for review), revealing the mosaic structure of genes caused by both transformation events and recombination. Detecting recombination requires sophisticated analysis (Sawyer, 1989; Stephens, 1985), but it does not seem unrealistic to presume these methods could be used on sequence data from hybrid zones. At present sequencing sections of the genome remains very time consuming, so an alternative might be to estimate the distribution of block sizes using closely linked markers. Consider two markers diagnostic of the rare immigrant alleles in a population such as that considered in this paper. Let

us assume that the distance between them, y is small enough to make the chance of two junctions occurring between them negligible. Then, if both indicate immigrant genetic material we may assume they lie on a single intact block, which by definition must be of length y or greater. If v(y) is the frequency of coupling haplotypes for markers y distant, then $v(y)=\int_y^1 p(z,t)dz$, where p(z,t) is the frequency due to $\Psi(z,t)$ blocks. Pairs of markers with different intervals can then be used to estimate p(z,t) for a number of points on the linear equilibrated portion of the distribution, and a number of points on the plateau in the distribution of very small blocks. This would be sufficient to allow an estimate of t as described in the present study. Such an approach might well be feasible where large numbers of markers are available: for example there are now of the order of 1000 micro-satellite markers for the *Mus* genome (Hearne, Ghosh, & Todd, 1992).

# (4)
# Invasion of Genomes

4.0 Introduction ............................................................ 64
4.1 Hitch-Hiking in a Single Population ............................................................ 69
4.2 Hitch-Hiking in Spatially Structured Populations ............................................................ 73
4.3 Discussion ............................................................ 85

# 4.0 Introduction

This chapter was inspired by Kate Abernethy's study of Sika and Red deer in Scotland (Abernethy, 1994a; Abernethy, 1994b). Sika phenotypes have increased since introduction of a few individuals early this century. However standard invasion models (Hengeveld, 1994) cannot be used to predict their spread because the introduced deer have hybridised with the native red deer, making the definition of an invading individual unclear. It seems that invasion combined with hybridisation has attracted little analytical study. The current author and Abernethy decided to collaborate in an attempt to predict the consequences of Sika introduction in Scotland, using likelihood analysis to fit models of invasion to Abernethy's data. This chapter reviews the issues involved in invasion with hybridisation, and develops and tests simple models of this process.

Invasion of introduced exotic organisms is an area of increasing ecological concern. The areas of interest have been categorised as (1) predicting which species will become invaders. (2) the way in which invasion progresses through space. (3) prediction of the rate of invasion. (4) the local build-up of the newly settled population. (5) the pheno-genetic differentiation of the new range. (6) the effect the invader has on native species. (Hengeveld, 1994). Categories 2,3 and 4 require modelling the invasion process, either mathematically or through explicit

simulation. This field of modelling stems from Fisher's (Fisher, 1937) classic paper describing the spatial spread of a favoured gene in a population using a non-linear reaction diffusion equation. Reaction terms are used to describe the increase of some element, while diffusion terms describe its spread. Such equations are analytically attractive because the waves of propagation they describe settle into steady forms, the shape and speed of which can often be determined. The ensuing literature on biological applications of reaction diffusion equations is now vast (Britton, 1986) spanning epidemiology, neurology, ontogeny, population biology and ecology.

The application of reaction diffusion equations to ecology dates from Skellam (1951), and has proved a powerful predictive tool for biological invasions (Andow,Karieva,Levin, & Okubo, 1990), though unsuitable when long distance dispersal is dominant (Nichols & Hewitt, 1994; Turelli & Hoffmann, 1991). The success of the reaction diffusion process has led to its continued elaboration, for example to consider age-structured populations (Hengeveld, 1994). Yet, the current models consider increase of a single variable: the frequency of a novel advantageous mutation or individuals of an introduced species, and are therefore inapplicable to an increasingly recognised subset of biological invasions: those involving break up of the invading genome through hybridisation. Increase of a set of genes introduced into a population requires multivariate analysis. Novel introduced alleles are by definition initially in complete linkage disequilibrium, and epistasis is likely to maintain this association,

so evolution of allele frequencies at different loci will be interdependent, as discussed in Chapter 1. The invasion of a genome highlights how the gene and organismal viewpoints on evolution can both be inadequate.

The integrity of species is so central to current invasion modelling that it is assumed without question in the categorisation of the field listed above. Yet, aside from Abernethy's study, there is increasing evidence that invasion with hybridisation is widespread. After initial introduction and establishment, range expansion of an invading genome will take the form of a moving hybrid zone between the invading and native taxa. Moving hybrid zones formed by secondary contact between divergent populations can be thought of as a special case of genome invasion within two established populations.

The spread of African or Africanised bees in South America has provoked much popular interest (Hall, 1990; Hall & Muralidharan, 1989; Harrison & Hall, 1993; Page, 1989; Smith, 1991). A predominantly African population of bees is now established in an area which was occupied by European subspecies for more than 100 years. Hybridisation between the two has almost certainly played some role in their spread, though perhaps to a relatively small degree (Smith, 1991); recent studies have illuminated mechanisms of hybrid inviability (Harrison & Hall, 1993).

Workers in the relatively new field of conservation genetics are concerned by the threatened loss of rare taxa through hybridisation, a problem for a number of taxa in the duck family (Laurel Hanna, pers. comm). Often it is difficult to decide what needs conserved. Much controversy arose over the protection of 'red wolves' of north America. It now appears that the red wolf phenotype is the product of hybridisation between grey wolves and coyotes (Lehman,Eisenhawer,Hansen,mech,Petersen,Gogan, et al., 1991); it has been shown that coyote mtDNA has introgressed into the grey wolf population.

Population cage experiments with two poeciliid fish taxa, *Gambusia affinis* and *G. holbrooki*, (Scribner & Avise, 1994) show marked hybridisation resulting in consistent loss of *G. affinis* nuclear and cytoplasmic alleles, and suggest strong directional selection in favour of *holbrooki* genotypes. These finding are consistent with the observed dynamics of *Gambusia* hybridisation in nature (Scribner, 1993), where ecological conditions seem to regulate a series of moving hybrid zones.

There is a well characterised moving hybrid zone between newts of the genus *Triturus* in western France (Arntzen & Wallis, 1991), evidence of movement coming from surveys separated by about thirty years, and genetic relics of introgression, termed the 'footprints' of the hybrid zone. The comparison over time makes this study very powerful; concerted movement of hybrid zones may only be detectable over a period of decades, and it is possible that some of what are perceived to be stable hybrid

zones are actually subject to selective increase of one genome. Likewise changes in ecological conditions may favour one of two taxa in a stable hybrid zone. Selective increase of a genome is obviously an important issue, if we are to develop useful models of the process, we must consider the mechanisms involved in detail.

We will begin by considering the increase of a genome whose advantage is due to many loci of additive effect. Thus we will only consider interaction between loci due to their initial association, not through epistasis. We can distinguish between two classes of model at the outset: those assuming discrete locus effects, and infinitesimal models. For the discrete locus case we will make the simplifying assumption that alleles at all loci are advantageous or neutral. We might imagine that disadvantageous alleles would quickly be lost during the establishment of an invading population. Neutral alleles will then hitch-hike behind selected, an effect we can quantify analytically to some extent.

An analysis of this nature is likely to be complex, as it combines two relatively unexplored areas: the simultaneous selective increase of many genes, and hitch-hiking in spatially structured populations. Here we will try to quantify the effect of hitch-hiking in spatially structured populations due to the increase of a single allele by comparison with results for a single panmictic population. Later we will discuss the implications of considering many selected loci.

## 4.1 Hitch-Hiking in a Single Population

The following treatment of hitch-hiking in a single population is modified after (Maynard Smith & Haigh, 1974). Consider a beneficial allele P with effect s and no dominance, giving viabilities

| PP | PQ | QQ |
|---|---|---|
| 1+2s | 1+s | 1 |

We can express the increase in the frequency p of the allele in a large population at Hardy-Wienberg equilibrium with non-overlapping generations as

$$\Delta p = \frac{spq}{\overline{w}} \qquad (4.1)$$

Where the fitness of the population is $\overline{w} = 1 + 2sp$. A neutral allele U will increase when associated with the selected allele, such that

$$\Delta u = \frac{sD}{\overline{w}} \qquad (4.2)$$

Where D is the linkage disequilibrium between P and U, and u is the frequency of U. We can express the change in D due to selection on the beneficial allele, followed by recombination between the loci in question as

$$\Delta D = D\,(1-r)(1+\tfrac{s}{\overline{w}}(p-q+spq)) - D \qquad (4.3)$$

If we assume recombination is weak, and selection is so weak as to make no difference to average fitness of the population $\overline{w}$, equations (4.1-3) simplify, and can be scaled by $t^*=st$, giving

$$\frac{dp}{dt^*} = pq$$
$$\frac{du}{dt^*} = D \qquad (4.4)$$
$$\frac{dD}{dt^*} = -\frac{D}{\theta} - D(p-q)$$

Where $\theta$ = s/r is the coupling coefficient. This system of equations can be solved to give

$$\frac{du}{dt^*} = \frac{v_0 p_0 q_0 e^{-t^*\left(1+\frac{1}{\theta}\right)}}{\left(p_0 + q_o e^{-t^*}\right)^2} \qquad (4.5)$$

Where $p_0$ and $u_0$ are the initial frequencies of the selected and neutral alleles, and u+v=1. By considering the change of u with p instead of t, we get a weak selection approximation for the net change in the frequency of the hitch-hiking allele U

$$\Delta u = v_0 \left(\frac{p_0}{q_0}\right)^{\frac{1}{\theta}} \int_0^{\frac{q_0}{p_0}} \frac{y^{\frac{1}{\theta}} dy}{(1+y)^2} \qquad (4.6)$$

Thus under the assumption of weak selection and recombination, the net change in the frequency of a hitch-hiker for a given set of starting conditions depends only on the coupling $\theta$, though the time taken for this change to occur will be a function of s, as

$t=t^*/s$. If we assume that initially the selected and neutral alleles only occur together, then $p_0 = u_0$ and Eq. (4.6) simplifies to

$$\Delta u = u_0^{\frac{1}{\theta}} \int_0^{\frac{1}{u_0}-1} \frac{y^{\frac{1}{\theta}}\,dy}{(1+y)^2} \tag{4.7}$$

It is straightforward to iterate equations (4.1-3) and compare the net change in a hitch-hiker with this weak s,r approximation (4.7). (Figure 4.1).

We can see that the weak s,r approximation overestimates the effect due to hitch-hiking, especially for loosely linked loci. In addition $\theta=s/r$ will be small for loosely linked loci. For example if we consider an unlinked allele hitching on an allele of large advantage $s=0.1$, $\theta$ is only 0.2. The effect of hitching between an increasing cytoplasmic factor and a nuclear locus, by their nature unlinked, was considered analytically by (Turelli,Hoffmann, & McKechnie, 1992), and was found to be small unless the cytoplasmic factor increased rapidly.

We can again use comparison with iterative solutions to see how good the weak s,r approximation is for different starting frequencies $u_0$, holding $\theta$ constant. The relationship of $\Delta u$ to $u_0$ is simplest for $\theta=1$, as Eq. 4.7 simplifies to

$$\frac{\Delta u}{u_0} = u_0 - \ln(u_0) - 1 \tag{4.8}$$

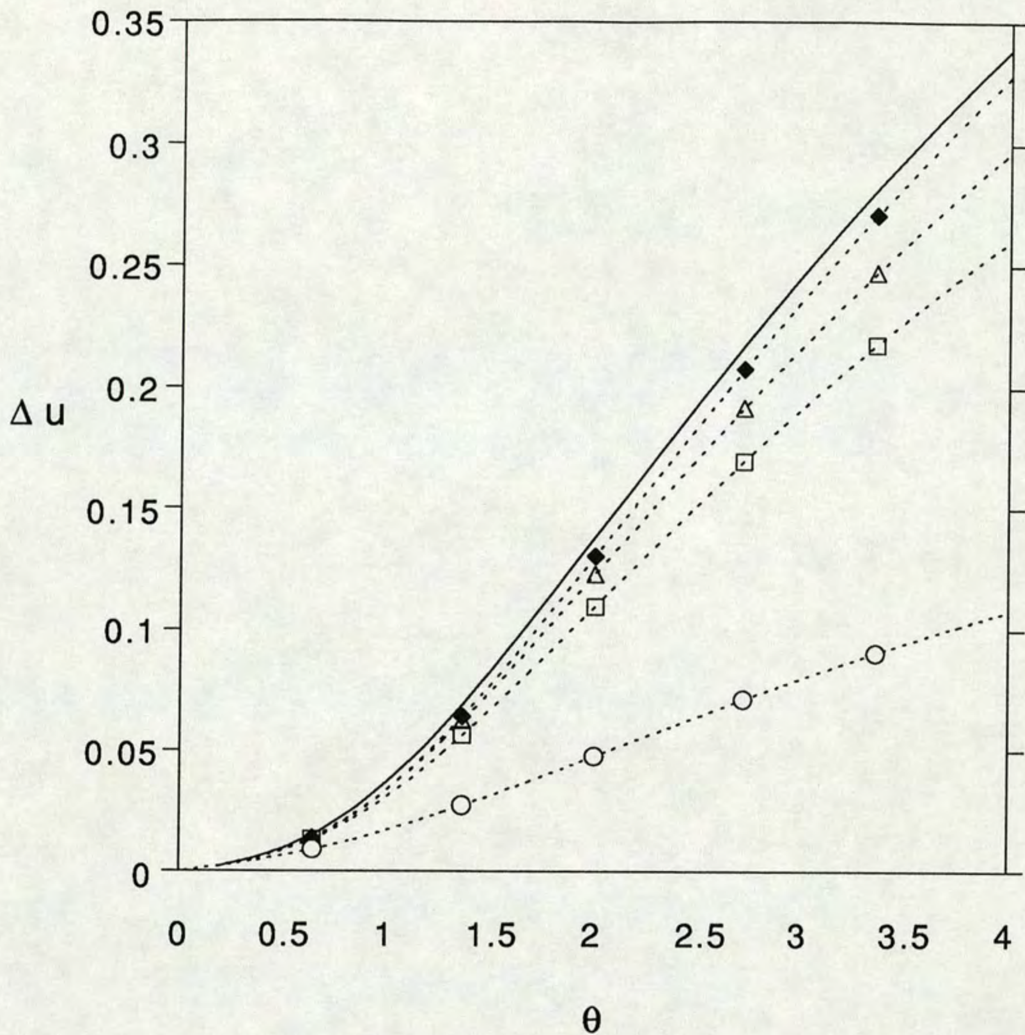



Figure 4.1: Comparison of the net change ($\Delta u$) in the frequency of a hitch-hiking allele initially at frequency $u_0=0.01$, with the coupling coefficient $\theta=s/r$. [solid line] the weak selection, recombination approximation (Eq. 4.7). [broken lines] exact solutions by iteration of equations (4.1-3). [Open symbols] hitchers at fixed map lengths: [circles] unlinked (r=0.5); [squares] r=0.1; [triangles] r=0.05. [Closed diamonds] hitchers at varying map distance from a selected allele of fixed advantage s=0.05.

Figure 4.2 compares Eq. (4.8) to iterative solutions of Eqs. 4.1-3 for $\theta=1$.

## 4.2 Hitch-Hiking in Spatially Structured Populations

The system of equations for a single population under weak selection (4.4) can easily be extended to the case of a one dimensional spatially structured population

$$\frac{dp}{dt^*} = pq + p'' \tag{4.9}$$

$$\frac{du}{dt^*} = D + u'' \tag{4.10}$$

$$\frac{dD}{dt^*} = -D\frac{1}{\theta} - D(p-q) + 2p'u' + D'' \tag{4.11}$$

Where primes denote the derivatives with respect to scaled distance $X = \sqrt{\frac{2s}{\sigma^2}}x$. Equation 4.9 is equivalent to Fisher's classic reaction diffusion equation (Fisher, 1937) for the spread of an advantageous allele, which after some period will form a travelling wave, the shape of which can then be solved numerically.

As yet no useful analytical solution of Fisher's travelling wave equation exists (though see Ablowitz and Zeppetella (1979) for an unhelpful solution), and the initial increase of the gene before a

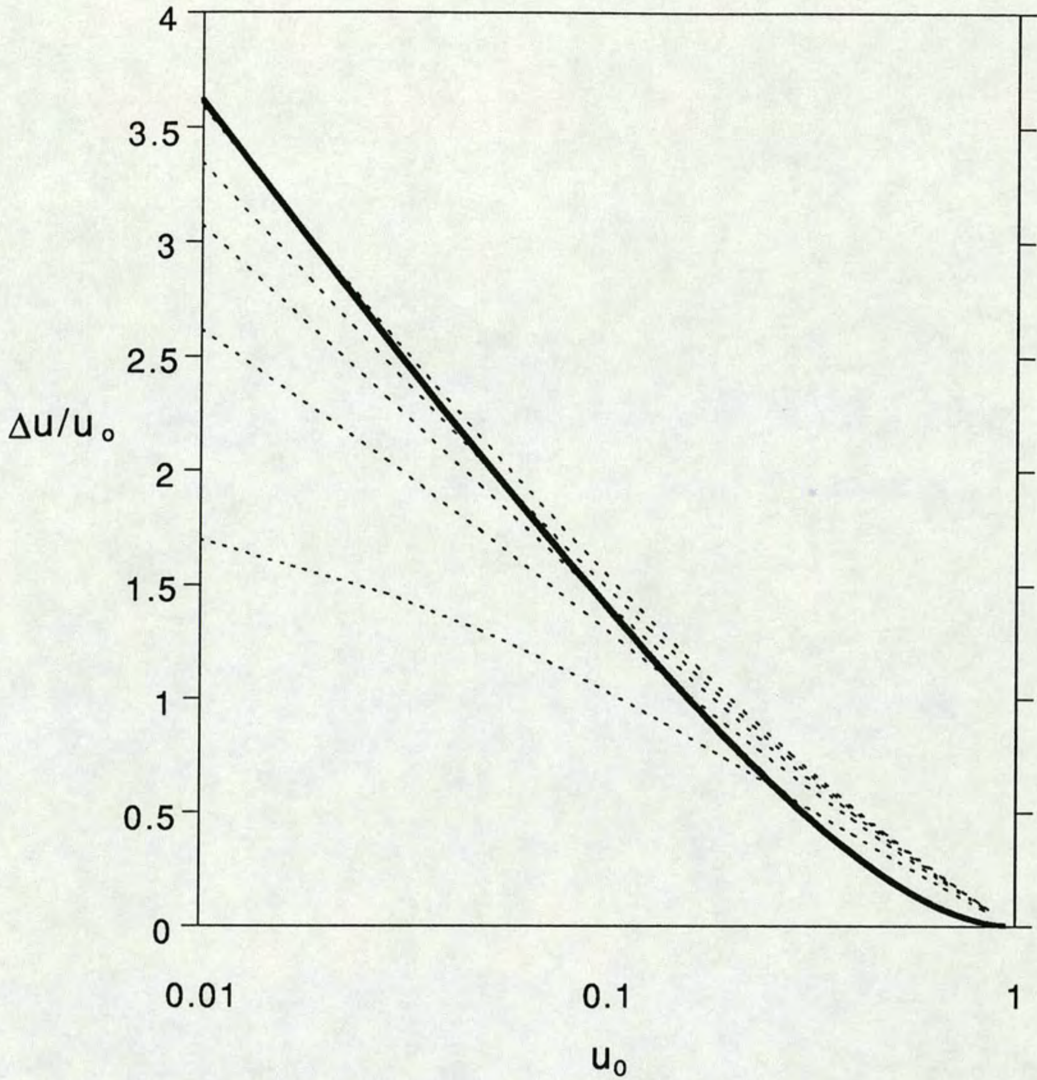



Figure 4.2: The relative increase ($\Delta u/u_0$) in the frequency of a hitch-hiking allele initially at frequency $u_0$ (shown on a natural log scale) for coupling $\theta=1$. [solid line] the weak selection, recombination approximation (Eq. 4.8). [broken lines] exact solutions by iteration of equations (4.1-3): Solution are for (from top to bottom) s=r=0.01, s=r=0.05, s=r=0.1, s=r=0.2, s=r=0.5

travelling wave is formed can only be predicted by iterating equation (4.7), or carrying out a simulation. Only under special conditions will D in Eq. (4.11) form a travelling wave. As there are no analytical solutions for the system of equations (4.9-11), it is useful to develop an alternative tool to study their behaviour.

We will develop a simple method to find iterative solutions for spatially structured populations, based on the 1D stepping stone model of populations (Kimura, 1953). A chain of demes with nearest-neighbour dispersal most closely approximates exchange across a continuum when they exchange half of their individuals, one quarter moving to each neighbour (Sawyer, 1976). In this case if the distance between neighbours is $\Delta x$, then the variance $\sigma^2$ of individuals' movements per generation is $\Delta x^2/2$. We can find iterative solutions equivalent to Eqs. (4.9-11) using a chain of demes, by following changes in p, u, and D within demes due to selection and recombination using equations (4.1-3), and then adjusting p, u and D in each deme to take into account dispersal between demes.

Initially selected and neutral alleles are introduced in complete association at frequency $u_{0,0}$ where $u_{i,t}$ is the value of u in deme i at time t. Equations (4.1-3) are used to find the values of $p_i$,$u_i$ and $D_i$ after selection and recombination. The value of $p_i$ in deme i>0 after dispersal is calculated as $p'_i = (p_{i-1}/4) + (p_i/2) + (p_{i+1}/4)$. We are modelling a single introduction into deme 0, with

symmetric migration, so for all demes i>0, $p_1 = p_{-1}$. Thus $p'_0 = (p_1/4) + (p_0/2) + (p_1/4)$, and we need not duplicate calculations for demes i<0. The values of $u_i$ and $C_i=D_i+p_i u_i$, the frequency of coupling gametes PU, are adjusted for dispersal in precisely the same way as $p_i$ for each deme. Finally, within each deme the value of D after dispersal is calculated as $D'=C'-p'u'$. The changes due to selection, recombination and dispersal are iterated in this way to give values for succeeding generations.

Although this approach is only an approximation to a continuous population, for weak selection it yields results for travelling waves in p very close to numerical solutions of Fisher's equation (Travelling waves are followed by discarding demes behind the point of fixation of p). This is true for both the velocity of the travelling wave and its shape (Figures 4.3, 4.4). As we would expect, travelling waves are slower, with a shorter wavelength than the weak selection approximation: the iterative solution allows for increase in average fitness in the population, which will slow the progress of the selected allele.

We can use this iterative method to estimate the increase in an allele hitching with a selected allele spreading in one dimension. In the absence of selection we would expect alleles to diffuse out from a point of introduction, with no net change in their numbers. A neutral allele hitching during the increase of a selected allele is pulled along behind its wave of increase (Figure 4.5), until recombination breaks down their association. At this point the neutral allele will continue to spread, but it will no

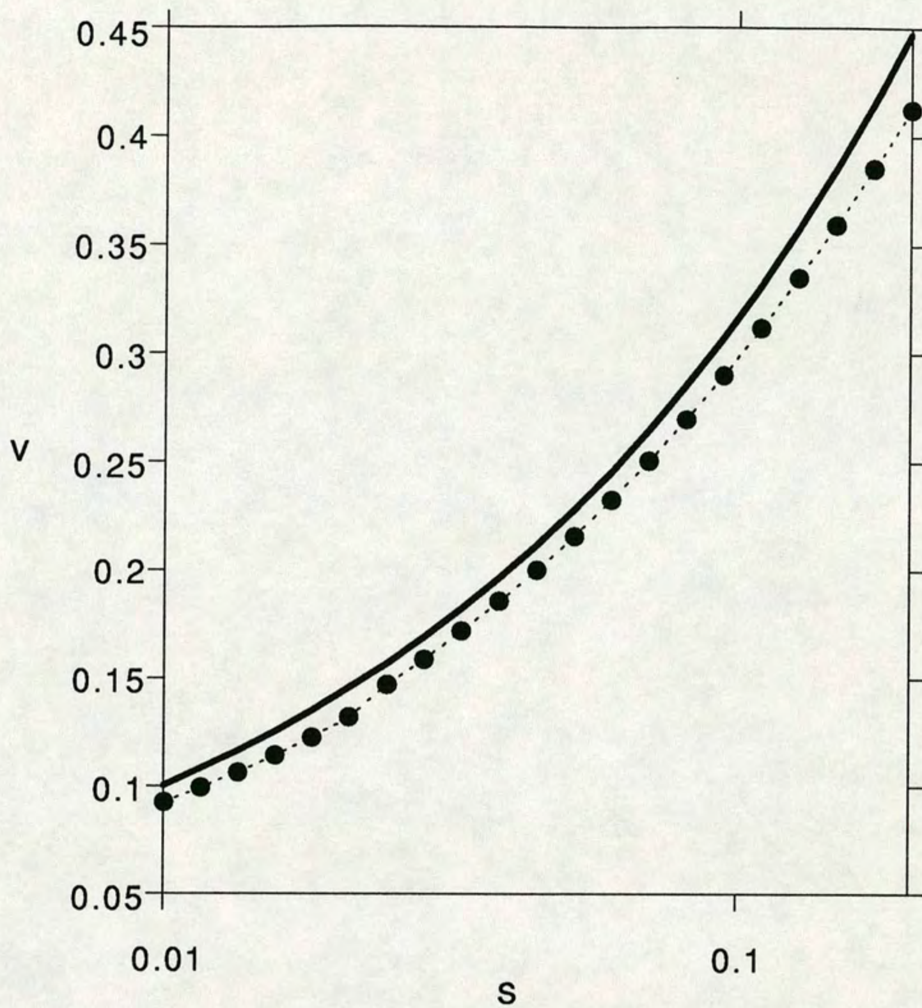



Figure 4.3: Velocities of iterated travelling waves [broken line] compared to the weak s,r Fisher wave expectation $v = 2\sqrt{\frac{s\sigma^2}{2}}$. As $\sigma^2=\Delta x^2/2$, $\Delta x$ is scaled to 1 giving expectation $v = \sqrt{s}$. s is shown on a natural log scale.

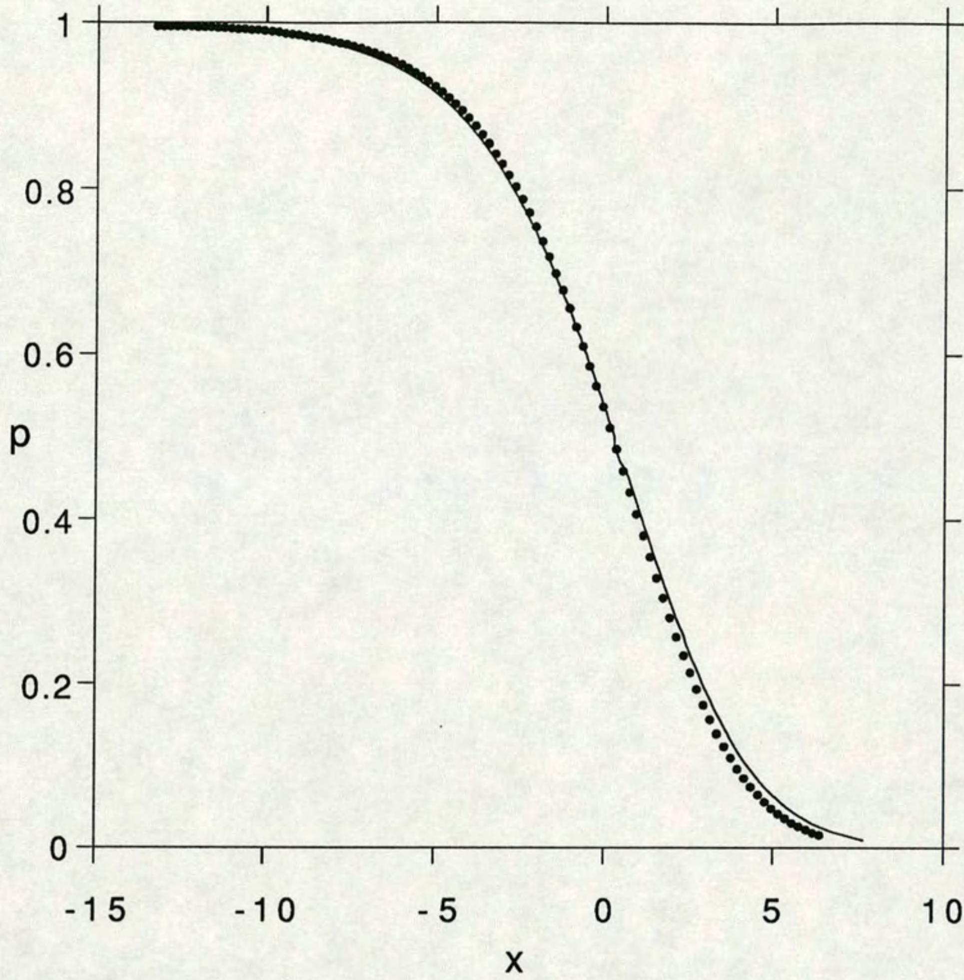



Figure 4.4: Shape of iterated travelling wave for s=0.01 [solid circles] compared to the weak s,r Fisher wave expectation ($\lambda$=5). The least squares fit gives $\lambda$=4.68 for the iterated data.

longer increase. The duration of the association between the alleles will determine the benefit the hitcher receives. The evolution of disequilibrium between selected and hitching alleles takes very different forms as selection is increased, even when the coupling is held constant. Figures 4.5 and 4.6 show the evolution of p, u, and D for coupling $\theta=1$, with different levels of selection and recombination. The qualitative difference in the evolution of disequilibrium for the same level of coupling suggests that the simple relationship between coupling and $\Delta u$ predicted by the weak s,r approximation (Eq. 4.7) will not hold for hitching in spatially structured populations.

Consider $D_t^*$ and $u_t^*$, the frequencies of $D_i$ and $u_i$ summed over all demes for our iterative model at generation t (demes $i<0$ are included in the summation, that is $u_t^* = u_{0,t} + 2\sum_{i=1} u_{i,t}$). With time, $D_t^*$ will tend to zero and $u_t^*$ will tend to a maximum value $\hat{u}^*$. Then $\Delta u^* = \hat{u}^* - u_{0,0}$ is a measure of the net gain in hitching alleles in the spatially structured population, and $\Delta u^*/u_{0,0}$ is precisely equivalent to $\Delta u/u_0$ for a single population. We must be more cautious in comparing $u_{0,0}$ and $\Delta u^*$ for a spatial population to $u_0$ and $\Delta u$ for a single population, as the spatial population variables must be scaled by the extent of the linear space being considered.

Figure 4.7 compares the relationship between $\Delta u^*$ (unscaled) and coupling for spatially structured populations to that for the single populations previously shown in Figure 4.1. It is clear that, excepting the trivial case of spread in a habitat of infinite extent, increased coupling has a greater effect on hitch-hiking alleles in a

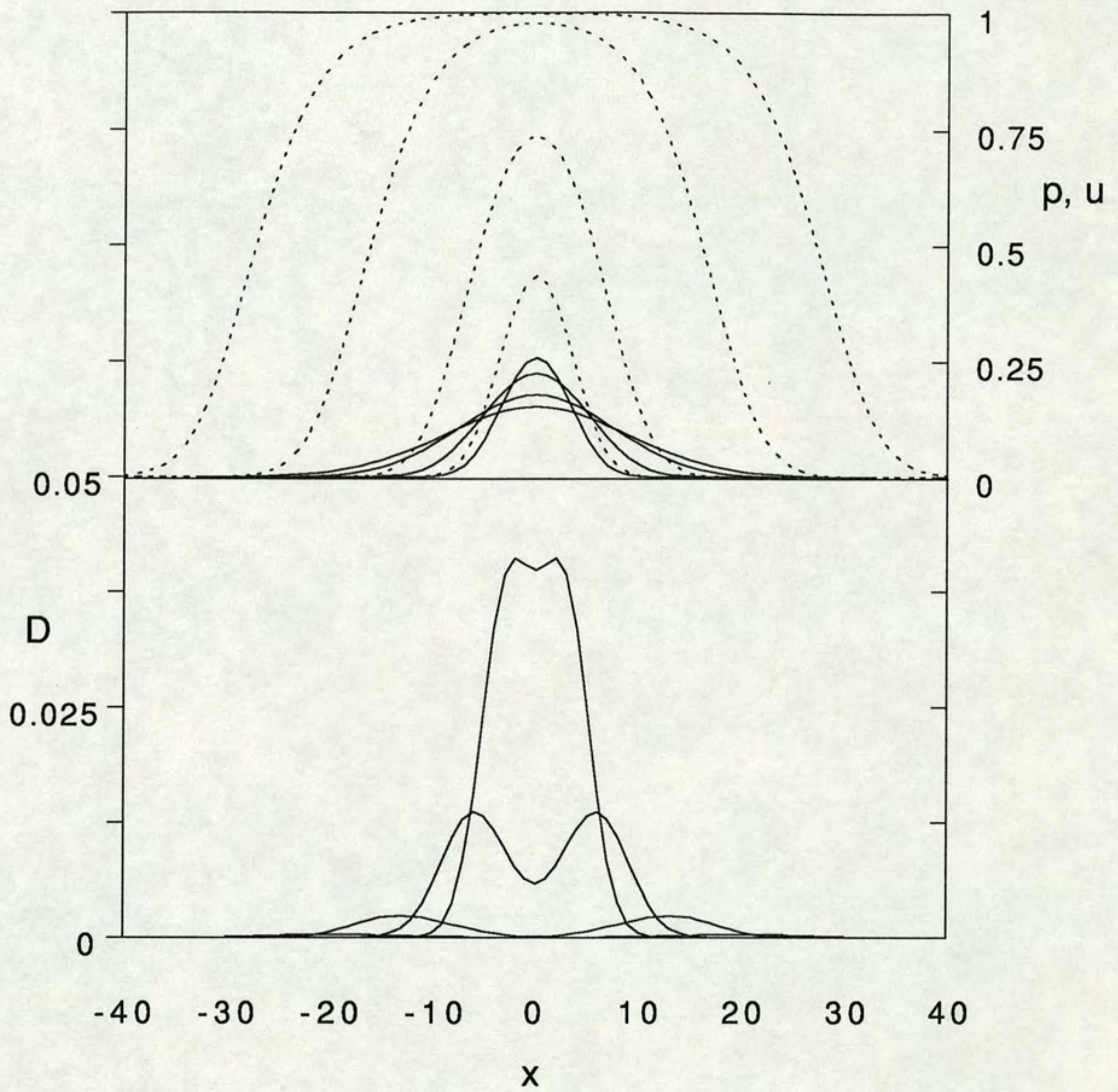



Figure 4.5: For coupling θ=1, s=r=0.1, $u_0$=0.01: the spread in one dimension of a selected allele frequency p [broken lines], a neutral hitcher frequency u [solid lines], and disequilibrium between them D [below], calculated by iteration over an array of demes. Curves are for t= 20, 40, 80 and 120 generations for each variable, and occur in that order from narrowest to widest.

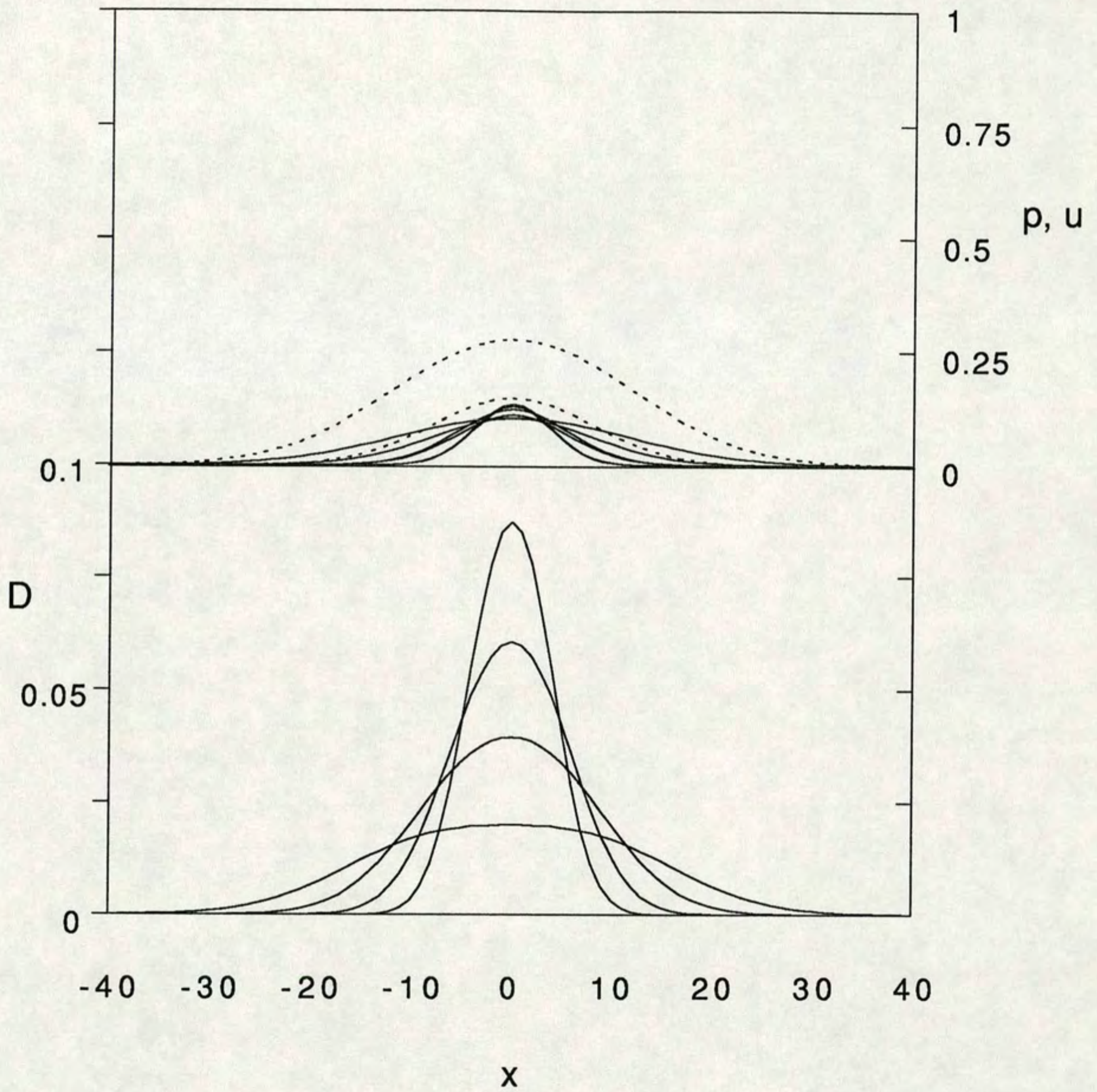



Figure 4.6: For coupling $\theta=1$, $s=r=0.01$, $u_0=0.01$: the spread in one dimension of a selected allele frequency p [broken lines], a neutral hitcher frequency u [solid lines], and disequilibrium between them D [below], calculated by iteration over an array of demes. Curves are for t= 30, 60, 120 and 240 generations for each variable, and occur in that order from narrowest to widest.

spatially structured population than in a single panmictic population. This holds even for loosely linked or unlinked hitching alleles. Figure 4.8 compares the relationship between $\Delta u^*/u_{0,0}$ and $\Delta u^*$ (unscaled) for spatially structured populations to that for the single populations previously shown in Figure 4.2. The curves for single and spatially structured populations are similar, and it seems likely that a suitable scaling of $\Delta u^*$ could make them coincident. If this is the case, then for a given coupling coefficient there is some characteristic extent of the linear population structure over which the relative increase of a hitch-hiking allele is identical to the relative increase of an allele in a panmictic population, for all initial frequencies. Figure 4.7 would then indicate that the 'characteristic extent' increases with increased coupling. This makes intuitive sense: in the limit of complete coupling (r=0) a hitching allele will spread forever.

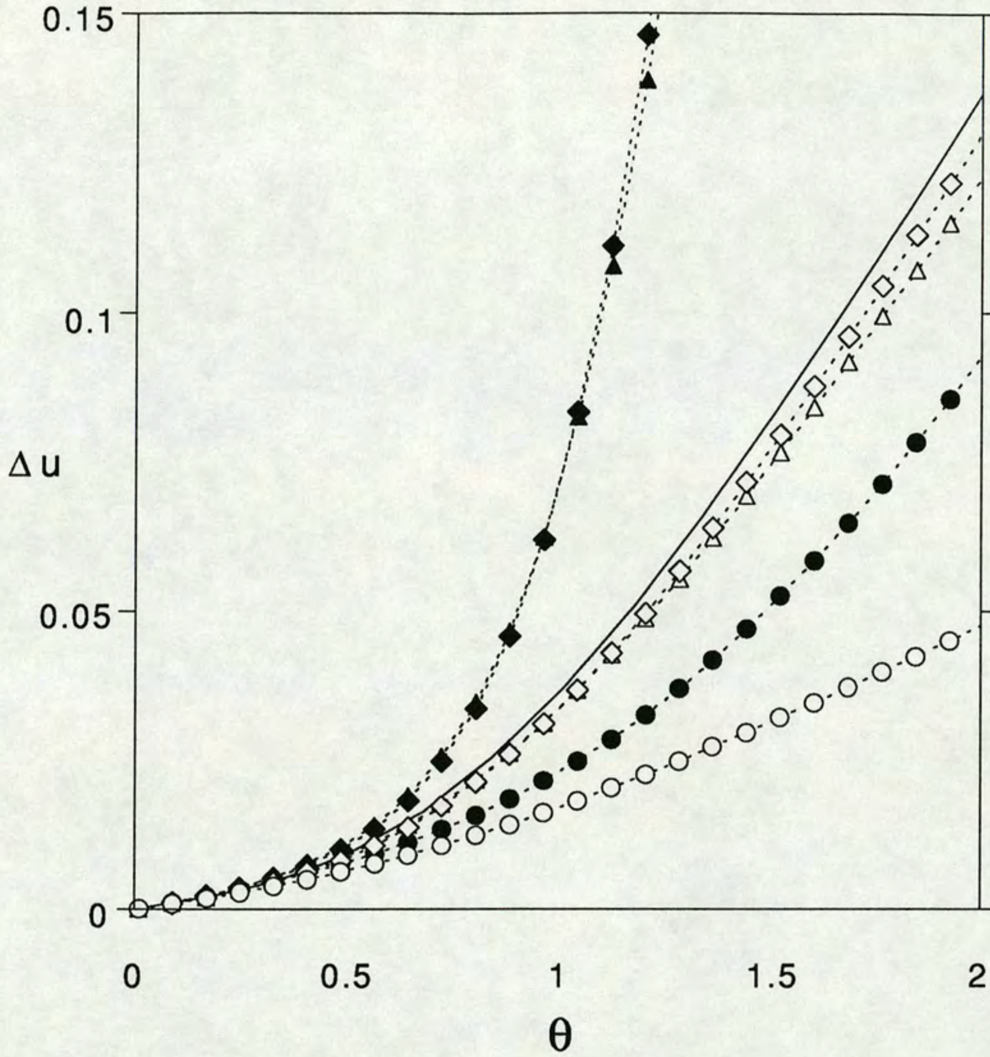



Figure 4.7: Comparison of the net change in the frequency of a hitch-hiking allele. Curves of $\Delta u^*$ for spatially structured populations [closed symbols] are overlaid on the curves for single populations [open symbols] shown in Figure 4.1. Initially $u_{0,0}=u_0=0.01$. [solid line] the weak selection, recombination approximation for a single population (Eq. 4.7). Hitchers at fixed map lengths: [circles] unlinked (r=0.5); [squares] r=0.1; [triangles] r=0.05. [diamonds] hitchers at varying map distance from a selected allele of fixed advantage s=0.05.

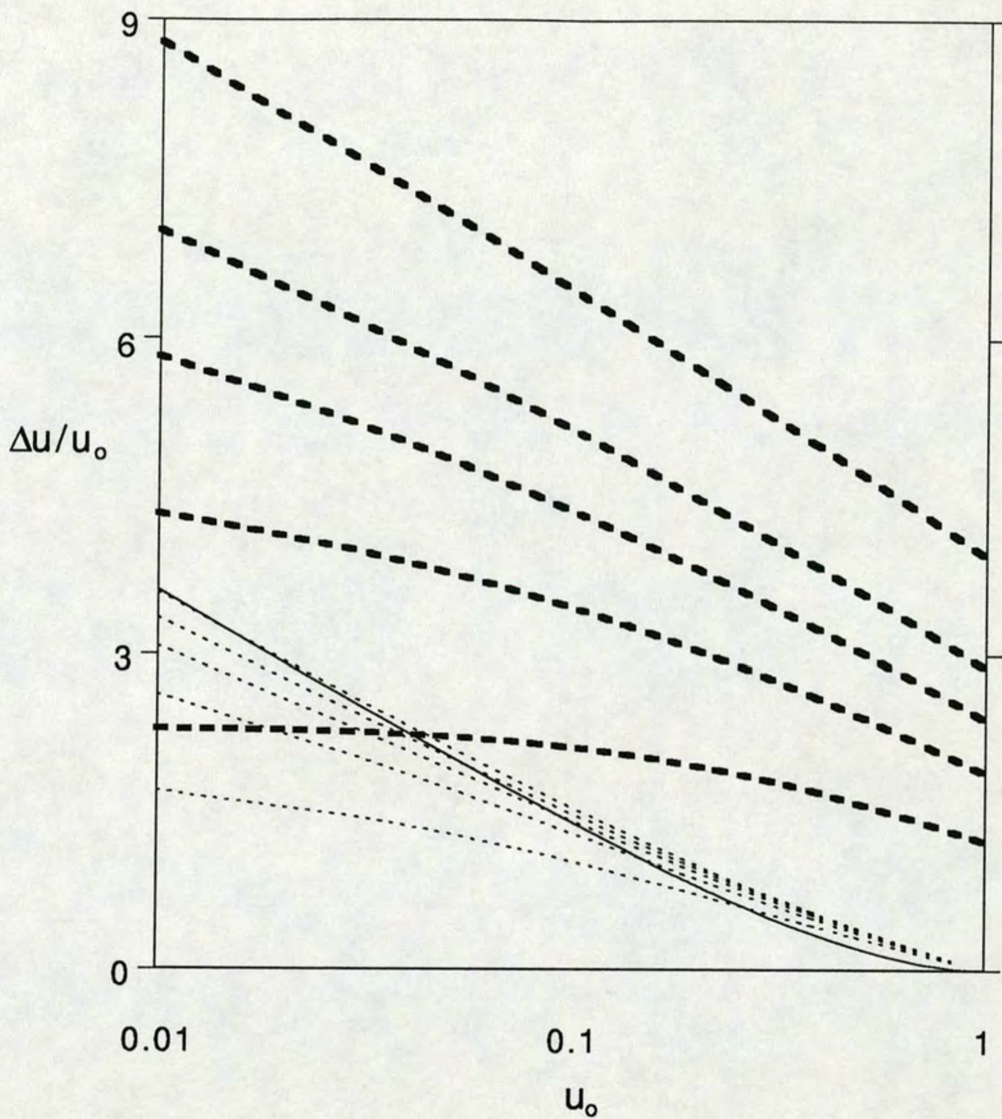



Figure 4.8: Comparison of relative increase in the frequency of a hitch-hiking allele, coupling $\theta=1$. Curves of $\Delta u^*/u_{0,0}$ vs. $u_{0,0}$ for spatially structured populations [bold dashed lines] are overlaid on the curves for single populations [broken lines] shown in Figure 4.2. [solid line] the weak selection, recombination approximation (Eq. 4.8). [broken lines] exact solutions by iteration of equations (4.1-3): Solutions are for (from top to bottom) s=r=0.01, s=r=0.05, s=r=0.1, s=r=0.2, s=r=0.5.

## 4.3 Discussion

We began by considering the processes involved in the increase of a genome whose advantage is due to many loci of additive effect, a complex problem, as it combines the simultaneous selective increase of many genes, and hitch-hiking in spatially structured populations. We have developed a framework for quantifying the effect of hitch-hiking in spatially structured populations due to the increase of a single allele. Extending this framework to consider many selected loci would be the next logical step.

Related problems have been studied for single panmictic populations. Ohta (1968) calculated the fixation probabilities of two selected loci in a finite population, and recognised that initial linkage disequilibrium would be important when considering crosses between divergent strains. Methods for mapping quantitative trait loci in selected strains through perturbation of frequencies neutral markers (Keightley & Bulfield, 1993) highlight the complexity of hitch-hiking during the increase of many loci.

It seems likely from these studies that mathematical analysis of hitch-hiking of many loci in a spatially structured population would be highly complex. As an alternative an infinite locus model can be developed from the finite population junctions model of Chapter 3, with the following modifications: The

population now consists of a string of finite demes. Demes exchange half of their individuals each generation, one quarter moves to each nearest neighbour, to approximate exchange across a continuum as discussed in section 4.2. Selection is additive across loci, and of the form outlined in section 4.1. A model of this type was tested for a hitch-hiking due to a single selected allele against the results developed in section 4.2, and then used in a likelihood analysis (Edwards, 1972) of data for neutral diagnostic sika markers on the Kintyre peninsula. The results of this analysis were encouraging, and are the subject of ongoing work.

# (5)
# Contribution to Future Generations

"My papers are my children."

(M. Turelli, pers. comm.)

5.0 Introduction........88
5.1 Branching Processes........89
5.2 The Junctions Model........92
5.3 Testing the Model........95
    5.3.1 Drift........95
    5.3.1 Recombination........96
5.4 Discussion........96

# 5.0 Introduction

There has been much popular interest in tracing ancestry of human populations. Studies of mitochondrial lineages avoid the problems which arises from sexual reproduction; recombination of the nuclear genome turns ancestral lineages from simple trees into complex nets. However, pinpointing a mitochondrial 'Eve' tells us little; extant human mitochondria must have a single common ancestor somewhere in time, but not necessarily in a human, and not necessarily in an individual which has contributed any other genetic material (Watterson & Donnelly, 1992). To gain useful resolution on population bottlenecks that might indicate cladogenesis we must consider many loci, with all the problems that entails. Here we consider one aspect of the many problems surrounding ancestry, the ultimate probability of fixation of a neutral block of genome.

The probability of ultimate fixation of a neutral allele under the Wright-Fisher model of random genetic drift is simply the initial frequency of the allele (Crow & Kimura, 1970). Thus a population size becomes large the probability of ultimate fixation tends to zero. This must be true of the alleles at each locus of an entire neutral genome. However, this genome may contribute alleles to many descendants during the many generations needed for a large population to fix. If the number of descendants *containing some contribution* from the ancestral genome becomes large relative to the population size, we might expect the probability of fixation of at least some of that genome to be significant. The

ultimate contribution of a neutral genome to a large but finite population does not seem immediately clear. This chapter addresses the problem by comparing junction simulations to analytical results based on a branching process argument developed by Barton.

A similar approach was used by Stam (1980) in deriving the distribution of the fraction of the genome identical by descent in finite random mating populations. Though related to fixation of the genome, this is a subtly different topic which is a generalisation of Fisher's (Fisher, 1953) work on inbreeding.

## 5.1 Branching Processes

Branching processes have been extensively used in the study of genetic problems (see Schaffer, 1970 for review). Barton has suggested application of the method from fixation of an allele to fixation of a genome. We shall first derive the single locus case, and then use the same form in deriving the result for fixation of a genome. Let the probability of ultimate fixation of a single copy of an allele in generation t, immediately before reproduction, be $P_t$. The number of genes in the population (2N) is assumed to be large enough that different alleles are lost independently of each other. The probabilities of fixation, $P_t$, can be found by iterating through one generation. The allele produces j offspring alleles with probability $\Psi_j$. This is the distribution of the number of heterozygotes produced by a rare heterozygote (which will

almost certainly mate with a homozygote for the common allele). Then:

$$(1 - P_{t-1}) = \sum_{j=0}^{\infty} \Psi_j (1 - P_t^*)^j \qquad (5.1)$$

Here $P_t^*$ is the probability that an allele at time (t-1) would be fixed, given that it is passed to precisely one offspring in the next generation, so by definition $P_t^* = P_t$ . If the distribution of offspring $\Psi_j$ is Poisson with mean (1+s), then:

$$(1 - P_{t-1}) = \sum_{j=0}^{\infty} \exp(-(1+s)) \frac{(1+s)^j}{j!} (1 - P_t)^j = \exp(-(1+s)P_t) \qquad (5.2)$$

For small s, this has the solution P=2s at equilibrium, so that the probability of ultimate fixation for a neutral allele is zero. The assumption of independent loss of alleles is equivalent to considering an infinite population under the Wright-Fisher drift process.

Now, consider a single block of genome, map length y at time t. Let the probability of ultimate fixation of at least some part of this block be P(y), immediately before reproduction. Each genome produces j offspring with probability $\Psi_j$ by mating with an unrelated individual. As with equation 5.1 for $P_t$:

$$(1 - P_{t-1}(y)) = \sum_{j=0}^{\infty} \Psi_j (1 - P_t^*(y))^j \qquad (5.3)$$

Here $P_t^*(y)$ is the probability that some part of a genome length y at time (t-1) would be fixed, given that it is passed to precisely one offspring in the next generation. With probability y there is exactly one crossover. This ensures that offspring inherit at most one block: With probability (1-y)/2, no block is passed on; with probability (1-y)/2, a block of map length y is passed on; and with probability y, a block of map length uniform in (0,y) is passed on. Then† :

$$P_t^*(y) = \frac{1-y}{2} P_t(y) + \int_0^y P_t(z)dz \tag{5.4}$$

In a population of steady size, neutral genomes must produce an average of two offspring, as Eq. 5.4 takes segregation into account. If the distribution of offspring is Poisson with mean 2, then:

$$(1 - P_{t-1}(y)) = \sum_{j=0}^{\infty} \exp(-2)\frac{2^j}{j!}(1 - P_t^*(y))^j = \exp(-2P_t^*(y)) \tag{5.5}$$

Substituting for $P_t^*(y)$:

$$P_{t-1}(y) = 1 - \exp\left[(1-y)P_t(y) + 2\int_0^y P_t(z)dz\right] \tag{5.6}$$

and $P_0(y) = 1$ by definition.

---

† This expression has a natural similarity to the change in the *frequency* of blocks size y due to recombination ( see terms in R, equation 2.1). Here genome length is succinctly expressed as map length (equivalent to Ry in 2.1), and the term for broken blocks considers descendants from, rather than ancestors of, blocks of length y.

The equilibrium solution for P(y) is not obvious but Eq. 5.6 can be iterated to find the solution at time t. A neutral genome is a collection of neutral alleles, and so the probability of ultimate fixation in an infinite population for any neutral genome P(y) must be zero. Eq. 5.6 does give a probability tending to zero, though very slowly. Mathematical analysis of this problem for finite populations is complex, as interactions between blocks must be considered. Here we aim to test the accuracy of the branching process as an approximation for finite populations. By comparing junction simulation results with iterative calculations of equation (5.6) we can determine whether interactions between blocks influences fixation in large finite populations.

## 5.2 The Junctions Model

We wish to simulate the break up and fixation of a finite population of N genomes in a diploid population under the action of drift and recombination. We will be comparing results with the branching process analysis above, so a model with discrete generations is appropriate. In the initial population we need to label each individual's genome as being distinct from all others, and we must then allow for junctions between N different haplotypes. When considering only two haplotypes as in the previous chapters, the state of a haploid after a junction is simply the alternative to the state before. To allow N states we have an additional variable associated with each junction which indicates the next state on the haploid (See Figure 5.1). This obviously

complicates the implementation of recombination, but the process does not change in essence. (Compare recombination in Appendix A1.2 and Appendix A2.0).

2N parents are drawn from the population at random; N pairings each produce one offspring to form the next generation of N individuals. It is impractical to simulate an infinite gamete pool; the consequences of the resultant genotypic sampling are considered when testing the model.

Recombination is uniform along the chromosome, with a number of chiasmata drawn from a Poisson distribution with expectation R, where R is the total map length and assuming no interference (See Appendix 1.1 for discussion). This is a more general model than the analytical description of recombination used above, which allows only one crossover within any homogeneous block; The models will be most similar when considering blocks of small map length.

The positions of all chiasmata for a chromosome pairing are drawn, ordered and then applied to the lists of junctions of the parents, to produce a single offspring which is added to the individuals of the next generation. The number of haplotypes remaining in the population to some degree was recorded each generation. Less frequently, the distribution of contributions of haplotypes to the population was estimated by counting the number of contributers in different classes. The breadth of each

n-state Haploid

Initial state=4

Junction Tuples: (x1,5) (x2,1) (x3,3)(x4,2)

(position,next state)

Haplotypes

1

2

3

4

5

Figure 5.1: Junction tuples used to represent a recombinant of many haplotypes.

contributing class was decreased exponentially from 2N, allowing fine resolution for smaller contributions. A haplotype is assigned to size class k if it contribution to the population is less than or equal to the $k^{th}$ class boundary, but greater than the $(k+1)^{th}$, where the $k^{th}$ boundary is $(7/8)^k$, and $40<=k<=0$. Thus size class zero runs from 2N to 2N.7/8, and so on. The number of haplotypes in a particular contribution class is then used as an estimator for the density of contributors at the mid-point of that class on a log scale.

## 5.3 Testing the Model

### 5.3.1 Drift

In the absence of recombination haploids will segregate in the same way as alleles at a single locus. If the model allowed an infinite gamete pool, drift in allele frequencies would be predicted by the Wright-Fisher model of gametic sampling. The genotypic sampling in the model should however be equivalent to gametic sampling when the population is in Hardy-Weinberg equilibrium. A diploid population of unique individuals with no selection will be in Hardy-Weinberg equilibrium in all but the first generation, when each individual is homozygous for its own (rare) alleles. Sampling in this case will be equivalent to sampling from a population of 2N distinct haploids. Using the matrix formulation of the Wright-Fisher model (after Hartl and Clarke 1989, p67), the state of the population with respect to alleles originating from an individual A can be described by the number

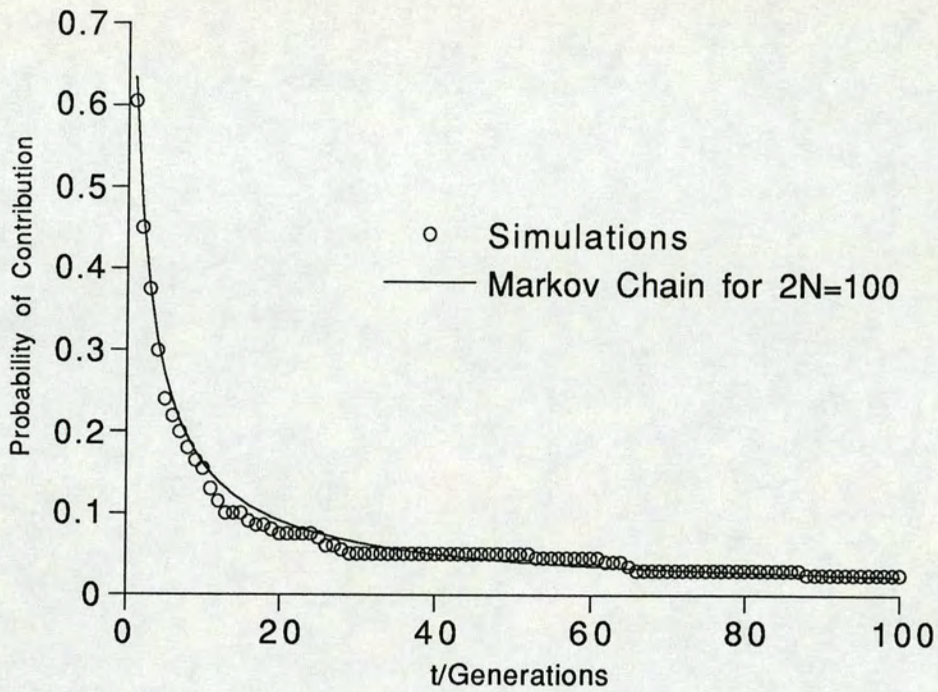
of 'A' alleles present after the first generation, with the population starting in state 1. Mathematica (Wolfram, 1992) was used to create the binomial probability transition matrix for a population 2N=100, and iterate a Markov chain starting with a vector in state 1. The probability that an individual will contribute some alleles to generation t is one minus the zero class after t iterations. When the Markov chain converges to a stationary distribution the probability of contribution for type A will equal its probability of fixation. If A hasn't fixed then some other type must have, and so A's contribution will be zero. Thus we expect the contribution to tend asymptotically to 1/2N, the initial frequency of the allele. There is a good between the drift expectation and simulation data (Figure 5.2).

### 5.3.1 Recombination

As all haplotypes are rare in the initial population, chiasmata will almost always result in junctions between different haplotypes. We would therefore expect recombination with map length R to produce on average R junctions per daughter haploid each generation, as long as no haplotype has increased to appreciable frequency through drift. Figure 5.3 compares the number of junctions per diploid individual for the initial generations of simulations with the expectation 2Rt.

## 5.4 Discussion

Figure 5.4 compares the number of haplotypes contributing to simulated populations of 2N=1000 with the branching process prediction for an infinite population (Eq. 5.6) which was iterated



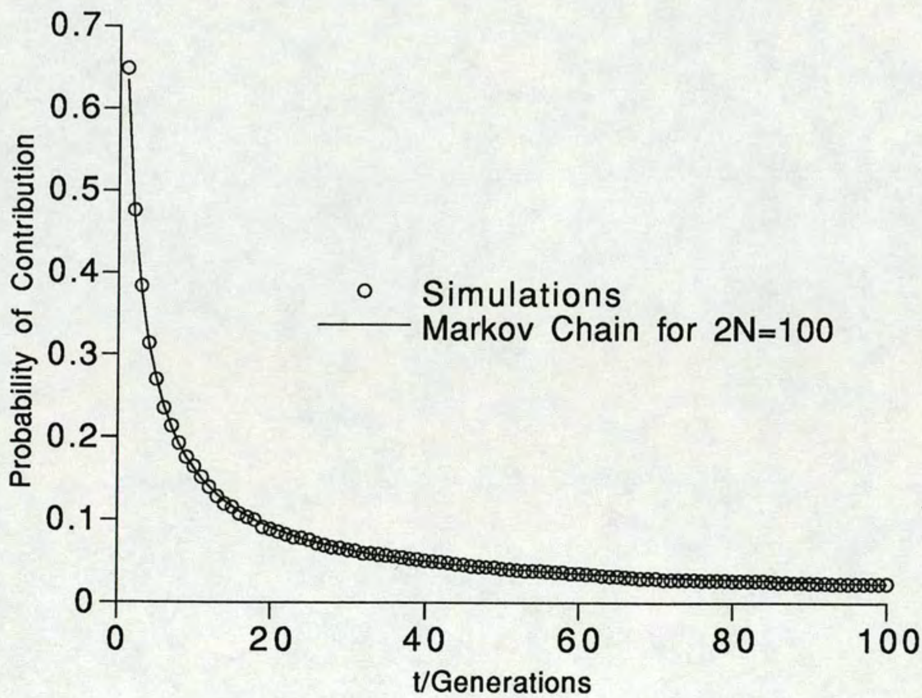
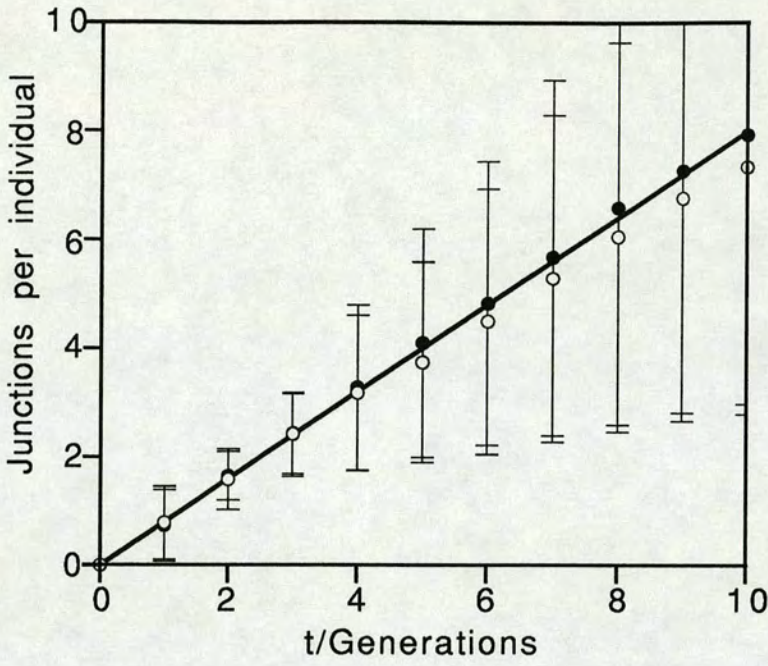




Figure 5.2: Comparison of Simulation results with the Fisher-Wright model of random genetic drift. Above: Simulation for 2N=100. Below Simulation for 2N=1000. Simulations are averaged over 2 replicates.



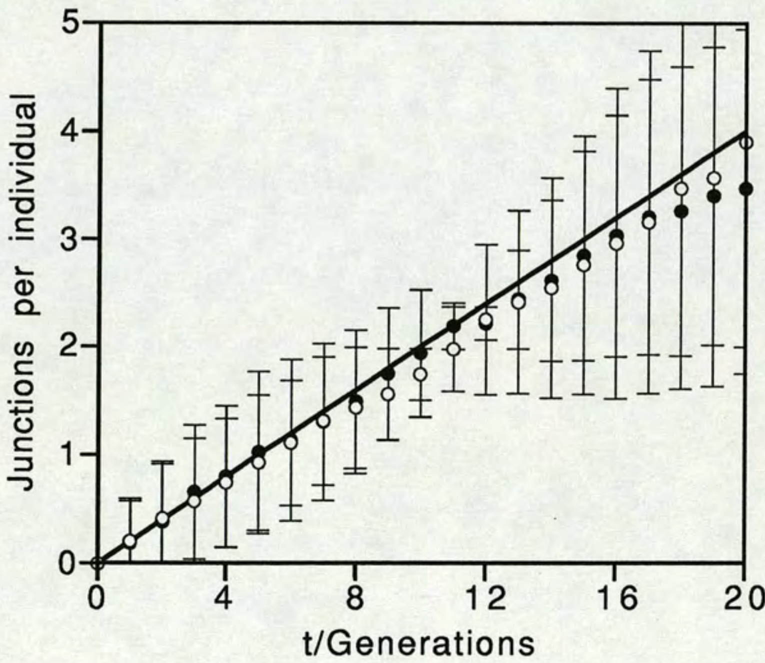




Figure 5.3: Testing the accumulation of junctions. Circles: Results for two simulations, with standard deviations. Solid line: The expectation 2Rt per diploid. Above R=0.4. Below R=0.1.

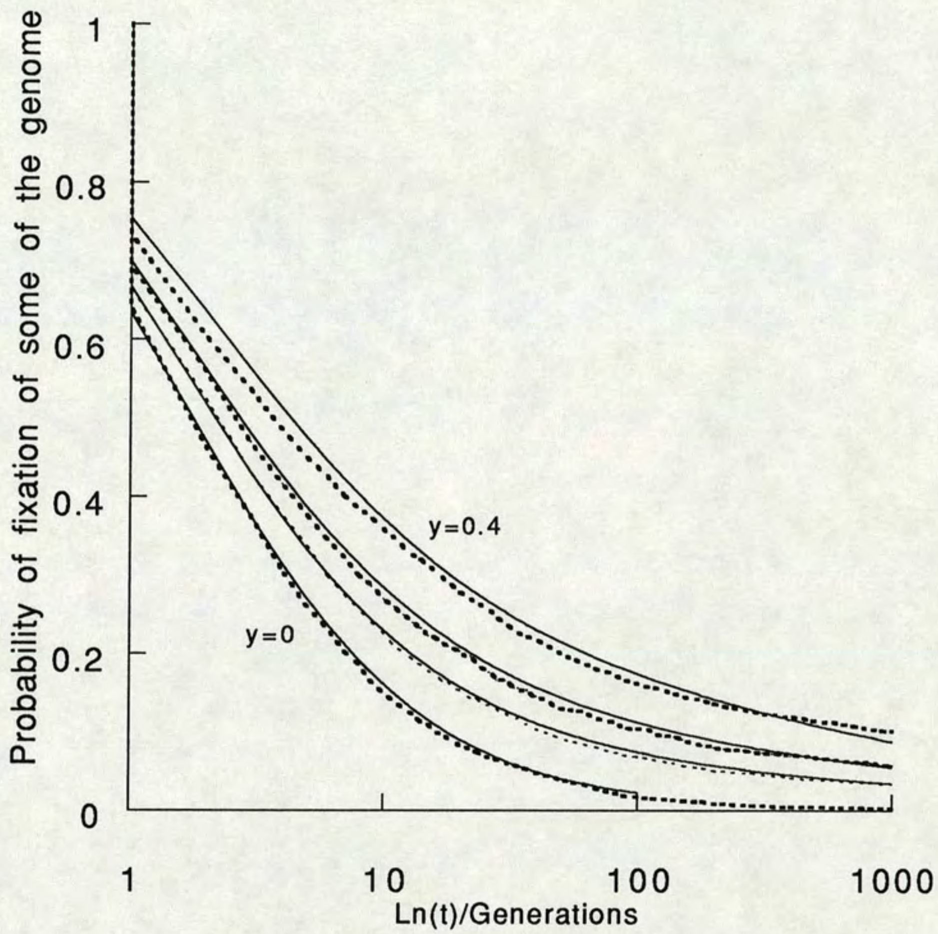



Figure 5.4: Simulation results [dotted lines] compared with, from top to bottom, iterative solutions of equation (5.6) for map length 0.4, 0.2 and 0.1, and a Markov chain solution for y=0.

using Mathematica (Wolfram, 1992). The simulation results correspond closely to the branching process prediction, indicating that descendants of a haplotype do not become sufficiently numerous relative to the population size to increase its chance of contribution. Figure 5.5 illustrates the distribution of the proportion of genome likely to be contributed by a haplotype for maplength 0.2, after 250 generations. The message is clear for individuals living in large populations: without selection or population growth, the only lasting contribution to future generations must be cultural!

Though the topic explored in this chapter is relatively simple, the simulation model developed may have potential for further applications. Recently Barton (1994) has described a general method for calculating the fixation probability of an allele which can find itself in a variety of genetic backgrounds, and has applied this method to find the effect of substitutions, fluctuating polymorphisms and deleterious mutations, though the model extends to cover any kind of population structure (Barton, 1993). This work, as above, uses the method of branching processes, and populations are assumed to be extremely large. Junction simulations seem a natural way to test the power of such results when applied to finite populations, and the model developed for the present study is in some ways pre-adapted for analysis of structured populations as it encorporates the functionality of the models used in previous chapters to study clines and hybrid zones.

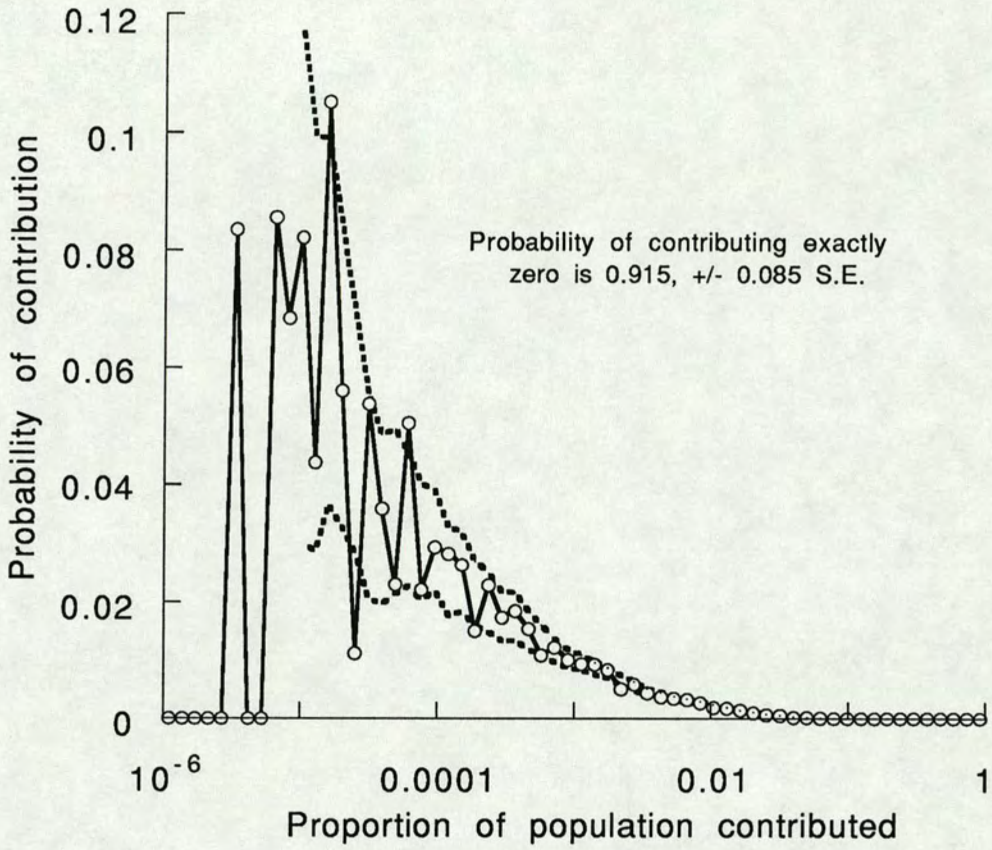



Figure 5.5: The distribution of contributions for map length y=0.2, after 250 generations. Results are averaged over 16 replicates. Dotted lines are 5-point running averages of the standard error.

# (6)
# Discussion

In the course of this thesis we have developed tools for simulation and analysis of multilocus problems. The junction approach has wide application, and can be used to extend existing work, and explore other areas of evolutionary biology.

One of the advantages of explicit simulation of junctions is that we are forced to consider the dynamics of populations as well as stable equilibrium states. This was highlighted in Chapter 2, where the simulations of multilocus clines revealed a very slow approach toward equilibrium. This prompted a mathematical analysis of the dynamics, and led to the method for recursive calculation of the upper bound on the distribution of blocks created as genotypes mix in the cline. The flux of blocks in a population due to selection, recombination and migration will often take a similar form to that considered in Chapter 2 (Eq. 2.9). For example we saw that the branching process expression for the probability of fixation of a block (Eq. 5.4) has similar terms for recombination, though here considering the flux into, rather than out, of a particular size class. If an equilibrium solution exists for an expression similar to Eq. 2.9 then the recursive method for finding solutions of the dynamics should be equally applicable, and as such this method may provide a general tool for the analysis of the dynamics of block distributions.

The method of dating hybrid zones described in Chapter 3 requires extension to consider exchange between a chain of demes, epistasis, and non-uniform selection over the genome, though this last may only be necessary when the number of

selected loci is small. In the case of a small number of selected loci, the distribution of blocks in hybrid genomes may be used to gauge the relative strength of selection at different loci, a method similar to the estimation of the effects of quantitative trait loci from the frequency of linked neutral markers (Keightley & Bulfield, 1993). Hybrid zones between *Mus* taxa exist throughout Europe, and a study of microsatellite markers has been carried out in the Pyrennees (Bonhomme, Catalan et al., 1984). The current author intends to use the models developed in Chapters 2 and 3 to assist interpretation of the data from this study.

The process of transformation in prokaryotes is central to the spread of resistance to antibiotics between clones (see Maynard Smith, 1990 for review). The relatively high ratio of mutation to transformation and recombination events in prokaryotes makes detection of junctions relatively straightforward. A junctions model of transformation among haploid populations could be used estimate the turnover of genetic material at blocks conferrinf resistance sweep to fixation.

Many problems in quantitative genetics lend themselves to the junctions approach. In particular, studies of the limits to artificial selection require consideration of many linked loci. Since Robertson (1970) it has generally been considered that for mass selection the selected proportion that maximises ultimate response is 1/2. Recently a simulation study (Hospital & Chevalet, 1993) suggested that with linkage the optimum selection intensity may be much lower than predicted unless population

size is small. From this study it seems that the distribution of selective effects among loci is critical. Simulation of many loci with non-uniform effects is inherently difficult. Referring back to our discussion of the junctions approach in Chapter 1, if the distribution of the effects of loci is complex, then a population will not initially consist of highly ordered haplotypes, and the junction description requires CL + 4NRt variables (Eq. 1.1) for a population of 2N haploids, with initially C distinct haplotypes of L discrete loci. Haploids in each generation can be described by reference to the original C haplotypes, and the positions of junctions between tracts of these haplotypes. When applied with care the junction approach is still more efficient than simply following all 2NL loci in the population. Robertson (1977) used a junction simulation to extend his work on artificial selection to the infinite locus case. In terms of the nomenclature above, he made reference to the original C haplotypes in the population using an elegant mechanism for the exact calculation of the variance associated with each new block produced by recombination. However, this method assumes a normal distribution of effects over loci. The current author, in collaboration with Frederic Hospital, is developing a junctions model which will directly reference a stochastically produced set of C original haplotypes, allowing any distribution of selective effects to be simulated. Progress in the mapping of quantitative trait loci (Keightley & Bulfield, 1993) indicates that the distribution of effects of loci is highly leptokurtic, so the junction model under development may be a useful tool for both theoretical and empirical study.

We have discussed applications of junction theory to quite specific problems. Although these studies are informative in their own right, they have a common thread in indicating the importance of linkage disequilibrium in natural populations. Analysing the dynamics of secondary contact hybrid zones, we have seen that alleles are likely remain associated to some extent over very many generations. We have seen that linkage disequilibrium between selected and neutral alleles may be slow to break down in spatially structured populations, increasing the hitch-hiking effect. By identifying the sources and degree of linkage disequilibrium in natural populations, we move closer to explanations of many of the larger issues in evolutionary biology, natural variation, recombination and sex, and speciation.

# ACKNOWLEDGEMENTS

I would like to thank Nick Barton for his enthusiasm and limitless patience.

I am indebited to my colleagues at ICAPB for many useful and encouraging discussions.

Without friendship and encouragement I would never have presented this thesis. Thankyou to many, especially

Alasdair Baird
Beate Nürnberger
Catriona Hayne
Catriona MacCallum
Christine Welch
Clive Lunny
David Baird
Dorothy Currie
Frederic Hospital
Gillian Baird
Gordon Mackenzie
Iain Baird
Isabelle Goldringer
Joel Peck
Julia Martin
Kevin Dawson
Kevin Fowler
Kirsty Vickers
Laurel Hanna
Lesley Smyth
Loeske Kruuk
Louise Graham
Rachel Baine
Rachel Taylor
Richard Hunter
Stuart Blackman
Susan Hughes
Samantha Holland
Timothy Graham
Kirsty Laughlin
Avis James

This work was funded by the Department of Education for Northern Ireland.

# REFERENCES


Abernethy (1994a) Doctoral Thesis, The University of Edinburgh.

Abernethy (1994b). The establishment of a hybrid zone between red and sika deer (Genus *Cervus*). Molecular Ecology, In press.

Ablowitz, M., & Zeppetella, A. (1979). Explicit solutions of Fisher's equation for a special wave speed. Bull.Math.Biol. , 41, 835-840.

Andow, D. A.,Karieva, P. M.,Levin, S. A., & Okubo, A. (1990). Spread of invading organisms. Landscape Ecology, 4, 177-188.

Arntzen, J. W., & Wallis, G. P. (1991). Restricted gene flow in a moving hybrid zone of the newts Triturus cristatus and T marmoratus in western France. Evolution , 45, 805-826.

Barton, N. H. (1983). Multilocus clines. Evolution, 37, 454-471.

Barton, N. H. (1993). The probability of fixation of a favoured allele in a subdivided population. Genet. Res. , 62, 149-158.

Barton, N. H. (1994). Linkage and the limits to natural selection. Genetics, Submitted.

Barton, N. H., & Gale, K. S. (1993). Ch. 2: Genetic Analysis of Hybrid Zones. In R. G. Harrison (Eds.), Hybrid Zones and the Evolutionary Process Oxford: University Press, Oxford.

Barton, N. H., & Hewitt, G. M. (1981). A chromosomal cline in the grasshopper *Podisima pedestris*. Evolution, 35, 1008-1018.

Barton, N. H., & Hewitt, G. M. (1985). Analysis of Hybrid Zones. Annual Review of Ecology and Systematics.

Barton, N. H., & Hewitt, G. M. (1989). Adaptation, Speciation and Hybrid Zones. Nature, 341, 497-503.

Berry, A. J.,Ajioka, J. W., & Kreitman, M. (1991). Lack of polymorphism on the *Drosophila* fourth chromosome resulting from selection. Genetics, 129, 1111-1117.

Bonhomme, F.,Catalan, J.,Britton-Davidian, J.,Chapman, V. M.,Moriwaki, K.,Nevo, E., & Thaler, L. (1984). Biochemical diversity and evolution in the genus Mus. Biochem.Genet. , 22, 275-303.

Britton, N. F. (1986). Reaction diffusion equations and their applications to biology. New York: Academic Press.

Bulmer, M. (1980). The Mathematical Theory of Quantitative Genetics. Oxford: Oxford University Press.

Clegg, M. T. (1978). Dynamics of correlated genetic systems II: Simulation studies of chromosomal segments under selection. Theoretical Population Biology, 13, 1-23.

Clegg, M. T.,Kidwell, J. F., & Horch, C. R. (1980). Dynamics of correlated genetic systems V: Rates of decay of linkage disequilibria in populations of *Drosophila melongaster*. Genetics, 94, 217-234.

Crow, J. F., & Kimura, M. (1970). An introduction to population genetics theory. New York: Harper & Row. New York.

Dawkins, R. (1982). The Extended Phenotype. Oxford: Oxford University Press.

Edwards, A. W. F. (1972). Likelihood. Cambridge: Cambridge University Press.

Felsenstein, J. (1988). Ch6: Sex and the evolution of recombination: An examination of current ideas. In R. E. Michod &

B. R. Levin (Eds.), The Evolution of Sex Sunderland, Massachusetts: Sinauer Associates Inc.

Fisher (1953). A fuller theory of junctions in inbreeding. Heredity, 8, 187-197.

Fisher, R. A. (1937). The wave of advance of advantageous genes. Ann. Eugenics , 7, 355-369.

Fisher, R. A. (1949). Chapter 3, Section 14. In The Theory of Inbreeding. (pp. 49-61). Edinburgh.: Oliver and Boyd.

Franklin, I. (1977). The distribution of the proportion of the genome which is homozygous by descent in inbred individuals. Theoretical Population Biology, 11, 60-80.

Franklin, I., & Lewontin, R. C. (1970). Is the gene the unit of selection? Genetics, 65, 707-734.

Futuyma, D. J. (1994). Ernst Mayr and evolutionary biology. Evolution, 48, 36-43.

Hall, G. H. (1990). Parental analysis of introgressive hybridization between African and European honeybees using nuclear DNA RFLPs. Genetics , 125, 611-621.

Hall, H. G., & Muralidharan, K. (1989). Evidence from mitochondrial DNA that African honey bees spread as continuous maternal lineages. Nature , 339, 211-213.

Harrison, J. F., & Hall, H. G. (1993). African-European honeybee hybrids have low non-intermediate metabolic capacities. Nature , 363, 258-260.

Harrison, R. G. (1993). Hybrid zones and the evolutionary process. Oxford: Oxford University Press.

Hartl, D. L., & Clark, A. G. (1989). Principles of Population Genetics (2nd ed.). Sunderland: Sinauer Associates, Inc.

Hearne, C. M.,Ghosh, S., & Todd, J. A. (1992). Microsatellites for linkage analysis of genetic traits. Trends in Genetics, 8, 288-294.

Hengeveld, R. (1994). Small-step invasion research. TREE, 9, 339-342.

Hill, W. G., & Robertson, A. (1966). The effect of linkage on limits to artificial selection. Genetical Research, 8, 269-294.

Hospital, F., & Chevalet, c. (1993). Effects of population size and linkage on optimal selection intensity. Theoretical and Applied Genetics, In Press.

Hunt, W. G., & Selander, R. K. (1973). Biochemical genetics of hybridisation in European house mice. Heredity, 31, 11-33.

Keightley, P. D., & Bulfield, G. (1993). Detection of quantitative trait loci from frequency changes of marker alleles under selection. Genet. Res. , 62, 195-204.

Kimura, M. (1953). Stepping stone model of population (Annual report No. 3). National Institute of Genetics of Japan.

Kondrashov, A. S. (1988). Deleterious mutations and the evolution of sexual reproduction. Nature, 336, 435-440.

Kreitman, M. (1987). Molecular population genetics. In Oxford Surveys in Evolutionary Biology Oxford: Oxford University Press.

Lehman, N.,Eisenhawer, A.,Hansen, K.,mech, L. D.,Petersen, R. O.,Gogan, P. J. P., & Wayne, R. K. (1991). Introgression of coyote mitochondrial DNA into sympatric North American gray wolf populations. Evolution , 45, 104-119.

Lewontin, R. C. (1974). The genetic basis of evolutionary change. New York: Columbia Univ. press.

Li, W. H., & Nei, M. (1974). Stable linkage disequilibrium without epistasis in subdivided populations. Theoretical Population Biology, 6, 173-183.

Maynard Smith, J., & Haigh, J. (1974). The hitch-hiking effect of a favourable gene. Genet.Res., 23, 23-35.

Moran, C.,Wilkinson, P., & Shaw, D. D. (1980). Allozyme variation across a narrow hybrid zone in the grasshopper *Caledia captiva*. Heredity, 44, 69-891.

Nichols, R. A., & Hewitt, G. M. (1994). The genetic consequences of long distance dispersal during colonization. Heredity , 72, 312-317.

Ohta, T. (1968). Effects of initial linkage disequilibrium on fixation probabilities for two loci. Ann.Rep.Nat.Inst.Genet.Jap. , 18, 67-67.

Page, R. E. (1989). Neotropical African bees. Nature , 339, 181-182.

Paige, K. N.,Capman, W. C., & Jennetten, P. (1991). Patterns across a cottonwood hybrid zone: cytonuclear disequilibria and hybrid zone dynamics. Evolution, 45, 1360-1370.

Robertson, A. (1970). Some optimal problems in individual selection. Theoretical Population Biology, 1(120-127).

Robertson, A. (1977). Artificial selection with a large number of linked loci. In International Conference on Quantitative Genetics, (pp. 16-21). Iowa State University: The Iowa State University Press / Ames.

Sawyer, S. (1976). Results for the stepping stone model for migration in population genetics. Ann. Prob. , 4, 699-728.

Sawyer, S. (1989). Statistical tests for detecting gene conversion. Mol. Biol. Evol., 6, 526-538.

Schaffer, H. E. (1970). Survival of mutant genes as a branching process. In Kojima KI, pp 317-336.

Scribner, K. T. (1993). Hybrid zone dynamics are influenced by genotype specific variation in life history traits: experimental evidence from hybridizing Gambusia species. Evolution, 47, 632-646.

Scribner, K. T., & Avise, J. C. (1994). Population cage experiments with a vertebrate: the temporal demography and cytonuclear genetics of hybridization in *Gambusia* fishes. Evolution, 48, 155-171.

Skellam, J. G. (1951). Random dispersal in theoretical populations. Biometrika , 38, 196-218.

Slatkin, M. (1975). Gene flow and selection in a two locus system. Genetics, 75, 787-802.

Smith, D. R. (1991). African honeybees in the Americas: insights from biogeography and genetics. Trends Ecol. & Evol. , 6, 17-22.

Stam, P. (1980). The distribution of the genome identical by descent in finite random mating populations. Genetical Research, 35, 131-156.

Stephens, J. C. (1985). Statistical methods of DNA sequence analysis: detection of intragenic recombination or gene conversion. Mol. Biol. Evol., 2, 539-556.

Szymura, J. M., & Barton, N. H. (1986). Genetic analysis of a hybrid zone between the fire-bellied toads, *Bombina bombina* and *B.variegata* near Cracow in Southern Poland. Evolution, 40, 1141-1159.

Szymura, J. M., & Barton, N. H. (1991). The genetic structure of the hybrid zone between the fire-bellied toads, *Bombina bombina* and *B.variegata* : Comparisons between transects and between loci. Evolution, 45, 237-261.

Turelli, M., & Hoffmann, A. A. (1991). Rapid spread of an inherited incompatibility factor in California Drosophila. Nature (ms).

Turelli, M.,Hoffmann, A. A., & McKechnie, S. W. (1992). Dynamics of cytoplasmic incompatibility and mtDNA variation in natural Drosophila simulans populations. Genetics , 132, 713-723.

Watterson, G. A., & Donnelly, P. (1992). Do Eve's alleles live on ? Genetical Research , 60, 221-234.

Wolfram, S. (1992). Mathematica: a system for doing mathematics by computer. In Redwood City: Addison-Wesley Publishing Company, Inc.

Appendix Contents

**Appendix 1**

**Implementation of Haploid Populations with Junctions Between Two States**

A1.0 Implementation Details and Overview .......... 2
A1.1 General unit .......... 12
implementation .......... 12
PickAreal .......... 15
tossAcoin .......... 15
uniform .......... 15
poisson .......... 16
A1.2 Junctions unit .......... 17
implementation .......... 20
RECOMBINE .......... 22
TRUE_STATE .......... 25
STATE_AT .......... 27
A1.3 Haploid unit .......... 32
implementation .......... 33
search .......... 34
transfer .......... 35
migrate .......... 36
immigrate .......... 36
double_immigrate .......... 37
infinite_immigrate .......... 37
A1.4 Stats unit .......... 38
implementation .......... 39
GET_CHIASMATA .......... 39
fitness .......... 40
getgenefreq .......... 40
find_wbar .......... 41
genefreqstats .......... 43
total_variance .......... 43
A1.5 Haploid Junctions program .......... 45
select .......... 46
parents .......... 46
reproduce .......... 47
parent .......... 48
infinite_reproduce .......... 48
initrep .......... 53
infinite_initrep .......... 54
update_stats .......... 54
get_seed .......... 56
main .......... 61

**Appendix 2**

**Inplementation of Diploid Populations with Junctions between N states**


A2.0 Junctions unit ........ 67
implementation ........ 72
RECOMBINE ........ 76
TRUE_STATE ........ 79
DOMINANCE ........ 82
HETEROZYGOSITY ........ 85
STATE_AT ........ 91
A2.1 Diploid unit ........ 96
implementation ........ 99
search ........ 100
transfer ........ 100
migrate ........ 101
adjustdemes ........ 102
immigrate ........ 103
double_immigrate ........ 103
infinite_immigrate ........ 104
A2.2 Stats unit ........ 105
implementation ........ 108
GET_CHROMOSOME_CHOICES ........ 108
GET_CHIASMATA ........ 108
fitness ........ 109
pointfitness ........ 110
getgenefreq ........ 111
getgenediseq ........ 111
find_extranuclear ........ 113
find_wbar ........ 113
A2.4 Diploid Junctions program ........ 117
infonlefttransfer ........ 118
advance ........ 119
select ........ 119
randomparent ........ 122
parent ........ 122
parents ........ 122
reproduce ........ 123
infinite_reproduce ........ 125
initrep ........ 131
infinite_initrep ........ 133
get_seed ........ 135
DO_OUTPUT ........ 144
MIGRATION ........ 148
main ........ 150

# Appendix 1
# Implementation of Haploid Populations with Junctions Between Two States

# A1.0 Implementation Details and Overview

Simulations using junctions appear throughout the preceding chapters. The details of their implementation are common to all applications.

Recombination is taken to be uniform along the chromosome, with a number of chiasmata drawn from a Poisson distribution with expectation R, where R is the total map length and assuming no interference. More complex distributions may be more realistic, and certainly would be easily implemented, but generally make comparison with accepted theoretical results difficult. It is important to note that the resolution of the numbers produced by the uniform random number generator used sets an upper limit on the number of loci which can be modelled. However, given the representation of real numbers on modern computers, and our expectation for the maximum number of junctions in a population ( see 1.1-3) this limit will be irrelevant for recombination over any realistic time scale. Data on the start state and junctions (See Fig. 1.1) of every haploid in the population are stored in each generation. Junctions are stored as 32-bit real numbers over the range 0 to 1, giving a maximum resolution of $\approx 10^8$ loci or base pairs per chromosome. The uniformity of the random number generator was checked.

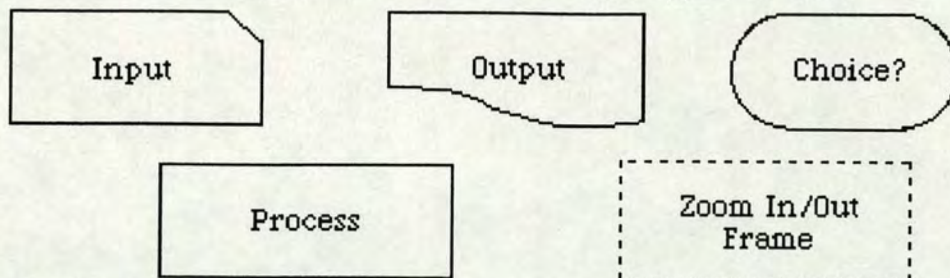
Simulation models were implemented in Think Pascal by Symantec, and run on several Apple Macintosh computers. The text of the implementation consists of a number of units of code, each covering a different topic, followed by the main program. The interface of a unit specifies those procedures and functions which can be used by other units and the main program. The implementation contains the body of the procedure listed in the interface, and other procedures necessary to that unit, but not to others.

## Overview

The purpose of the following diagrams is to present the simulation implementations in an intuitively accessible way. The diagrams are a series of recursive transition networks (RTNs) which serve as a map to key parts of the source code. These diagrams are not meant to be exhaustive - the shortest exhaustive description of the source code is the source code itself. Also the source code allows a number of situations to be simulated which bear no direct relevance to this thesis, such as sexually dimorphic populations, and genomes with many chromosomes. Such extraneous aspects are not referred to in the RTNs, instead they are designed to serve as a hierarchical overview of how the most important parts of the simulations function. At each level in the hierarchy more detailed levels are described by a description of their purpose, and a reference to the diagrams describing their function. The lowest levels refer to sections of the Pascal implementations for haploid populations with junctions between two state (Appendix 1.2) and of Diploid Populations with Junctions between N states (Appendix 2). The level at which descriptions change from diagrams to source code is approximately that at which the complexity or bulk of further recursive diagrams would exceed that of the source code.

The RTNs are intended to be taken together with the listing of all important functions in the Appendix Contents, as keys to the purpose and detailed implementation of the source code.

**RTN figures are made up of five types of unit**

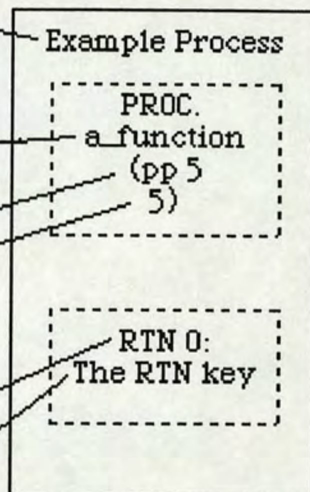


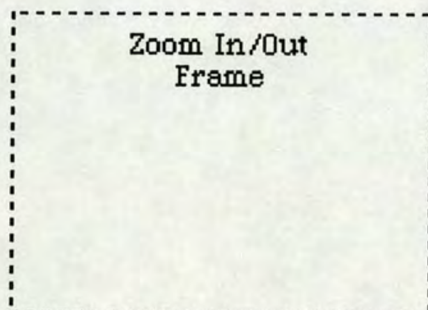

**Each process box is headed by its description**

**A process may contain references to source code implementations, denoted by PROC.**
**Such references are followed by the relevant page numbers in appendix 1 and appendix 2**

**A process may contain references to another RTN diagram, denoted by RTN**

**Such references are followed by the RTN title**




Sublevel of RTN 0.

**The Zoom In/Out Frame indicates a unit which appears on other levels of the hierarchical description.**

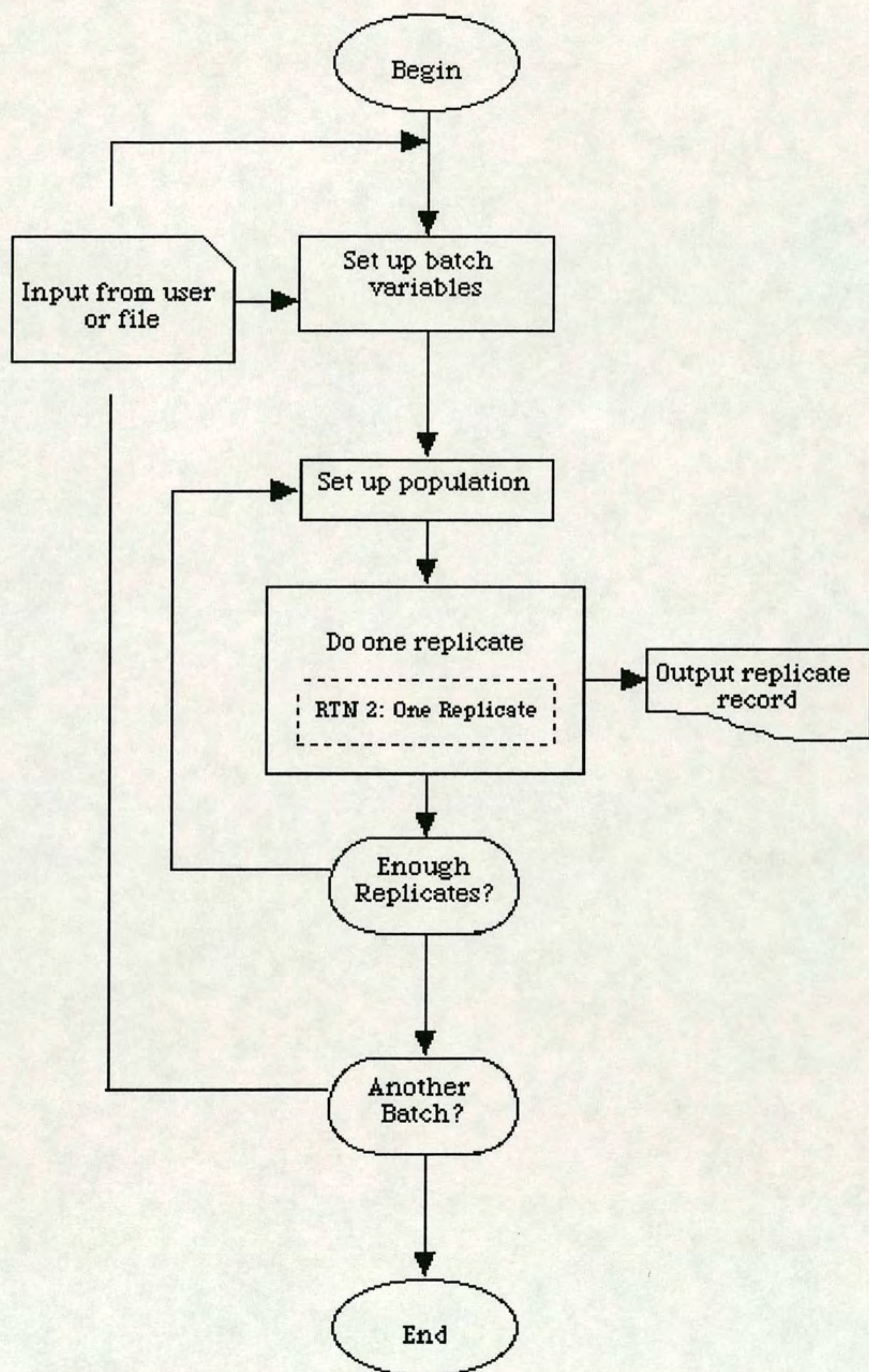
**Where higher levels of description are not obvious, they are indicated outside the bottom left corner of the frame.**

**The title of each RTN is followed by the page numbers of the relevant source code in each implementation respectively.**

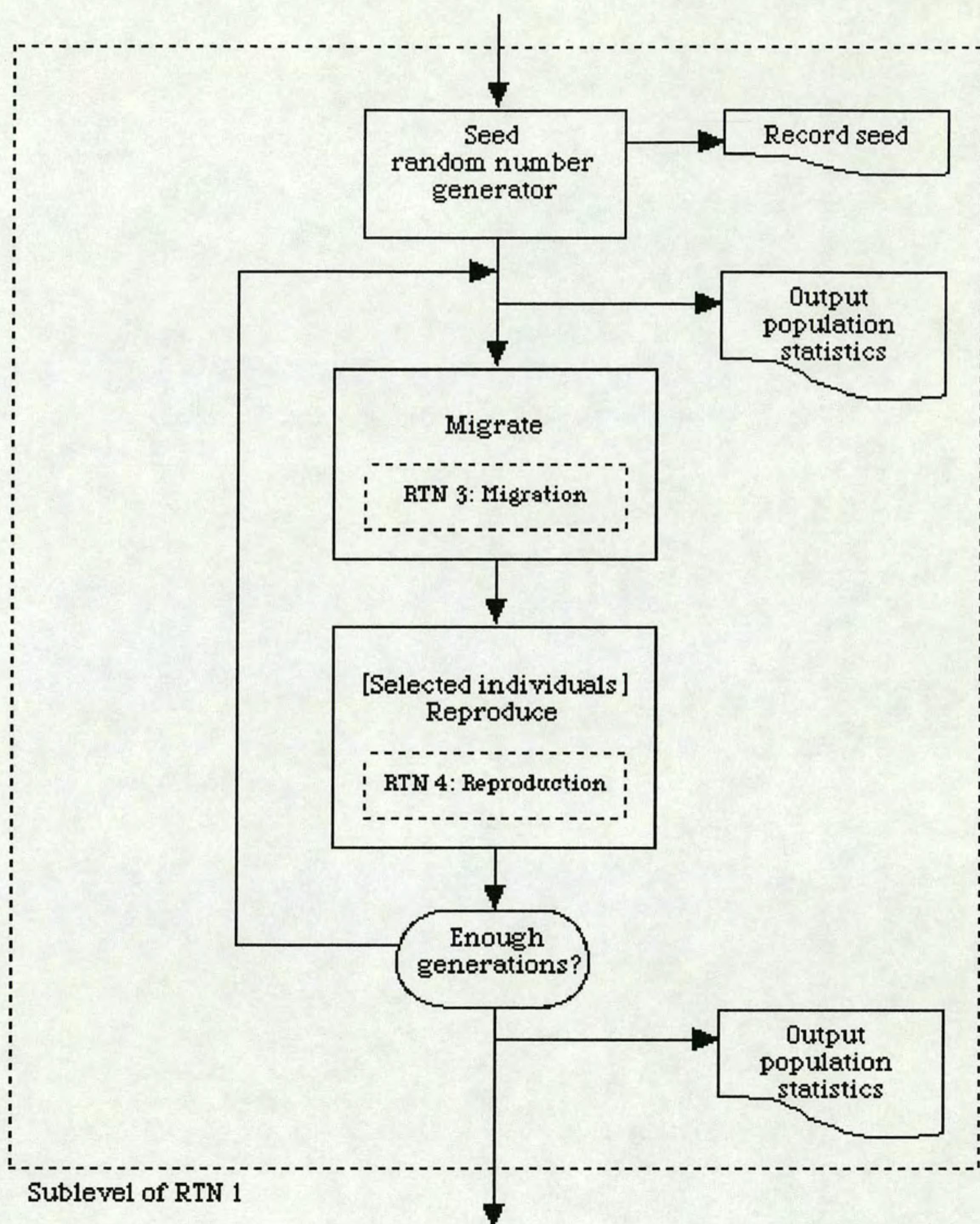
**(In addition the source code contains references back to relevant RTNs.)**

```
function a_function:string; {[ RTN 0, p5]}
begin
end;
```

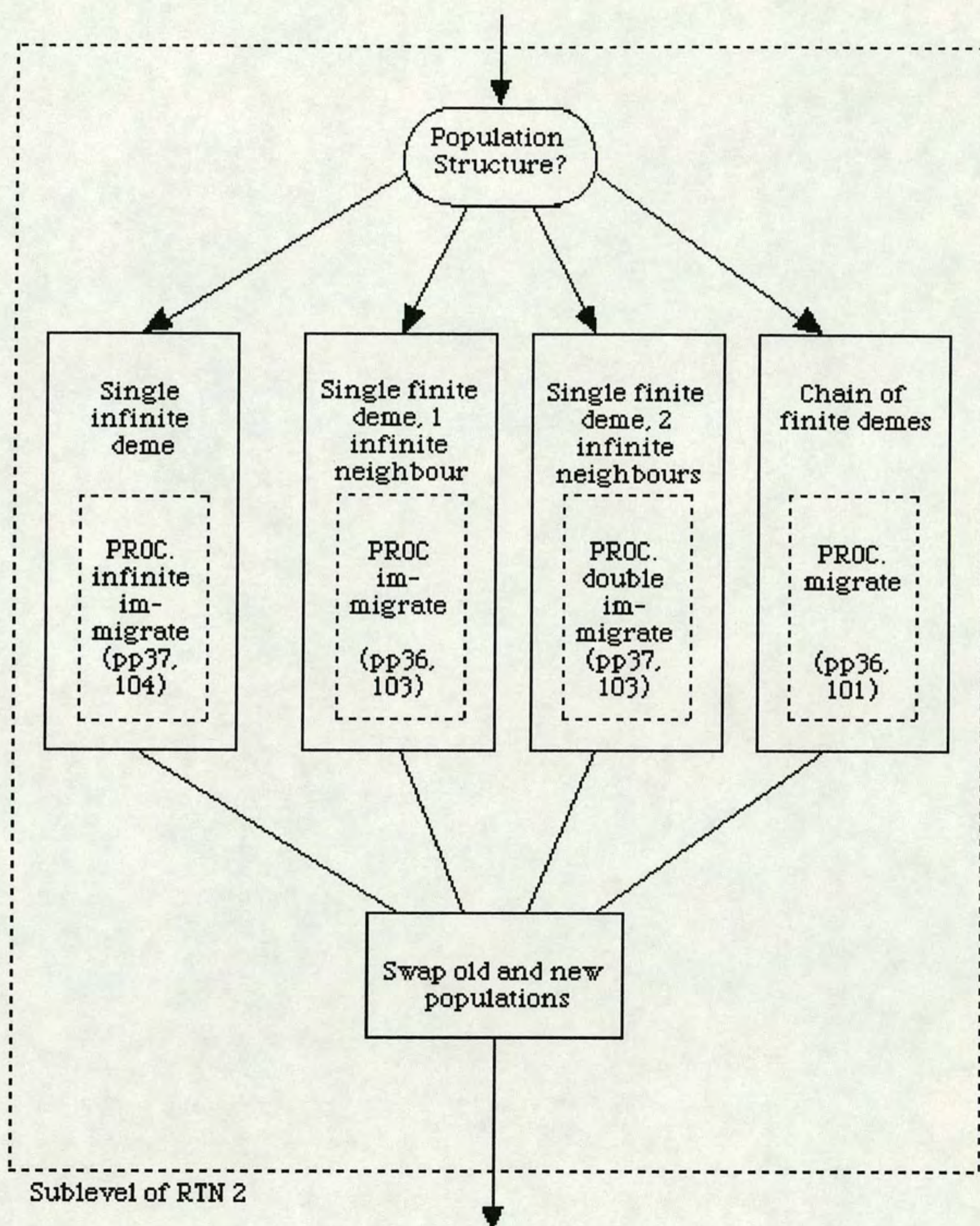
RTN 0 **(Not an RTN)** :A Key to RTN diagrams (pp5, 5)



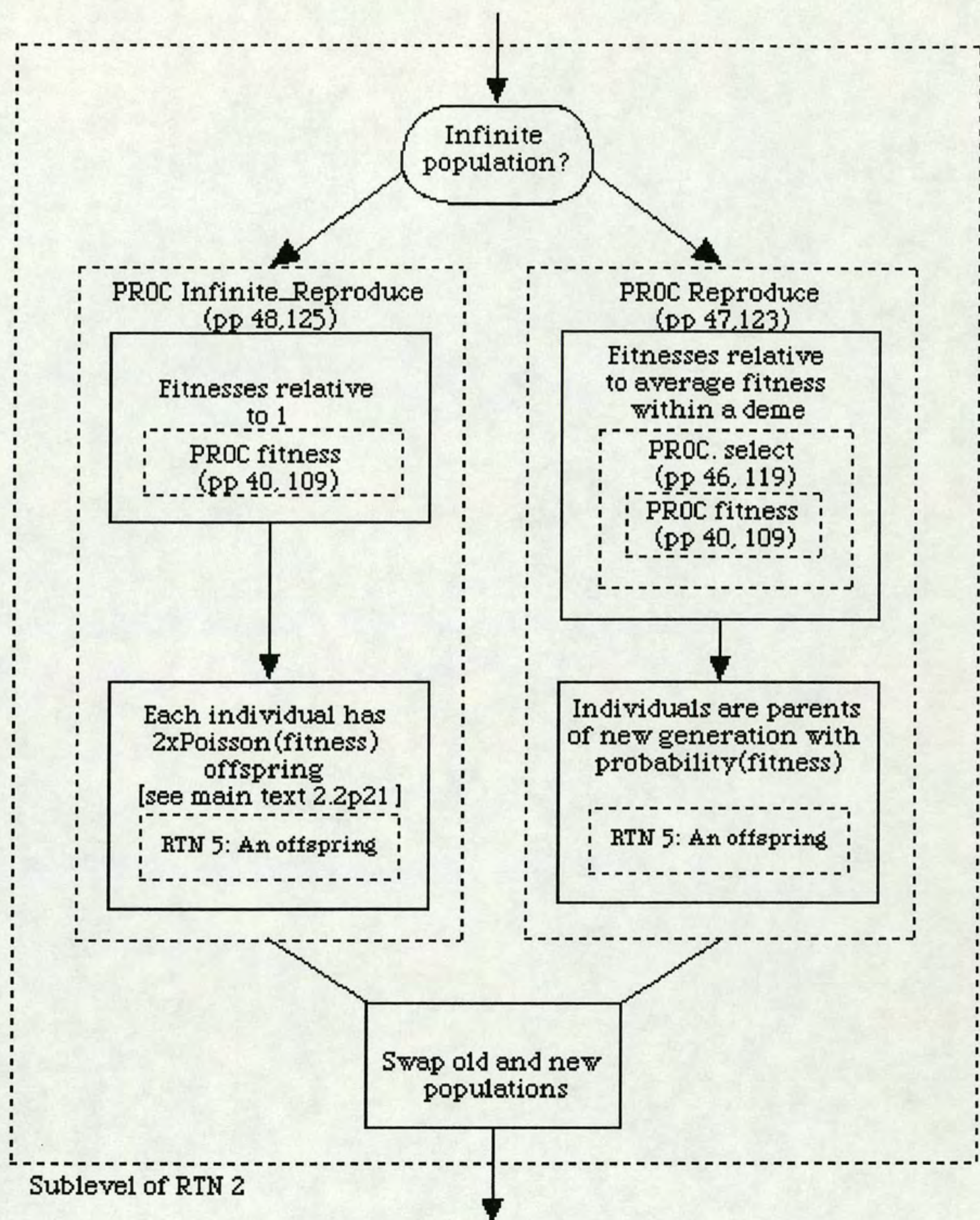

RTN 1: The Entire Program (pp61, 150)



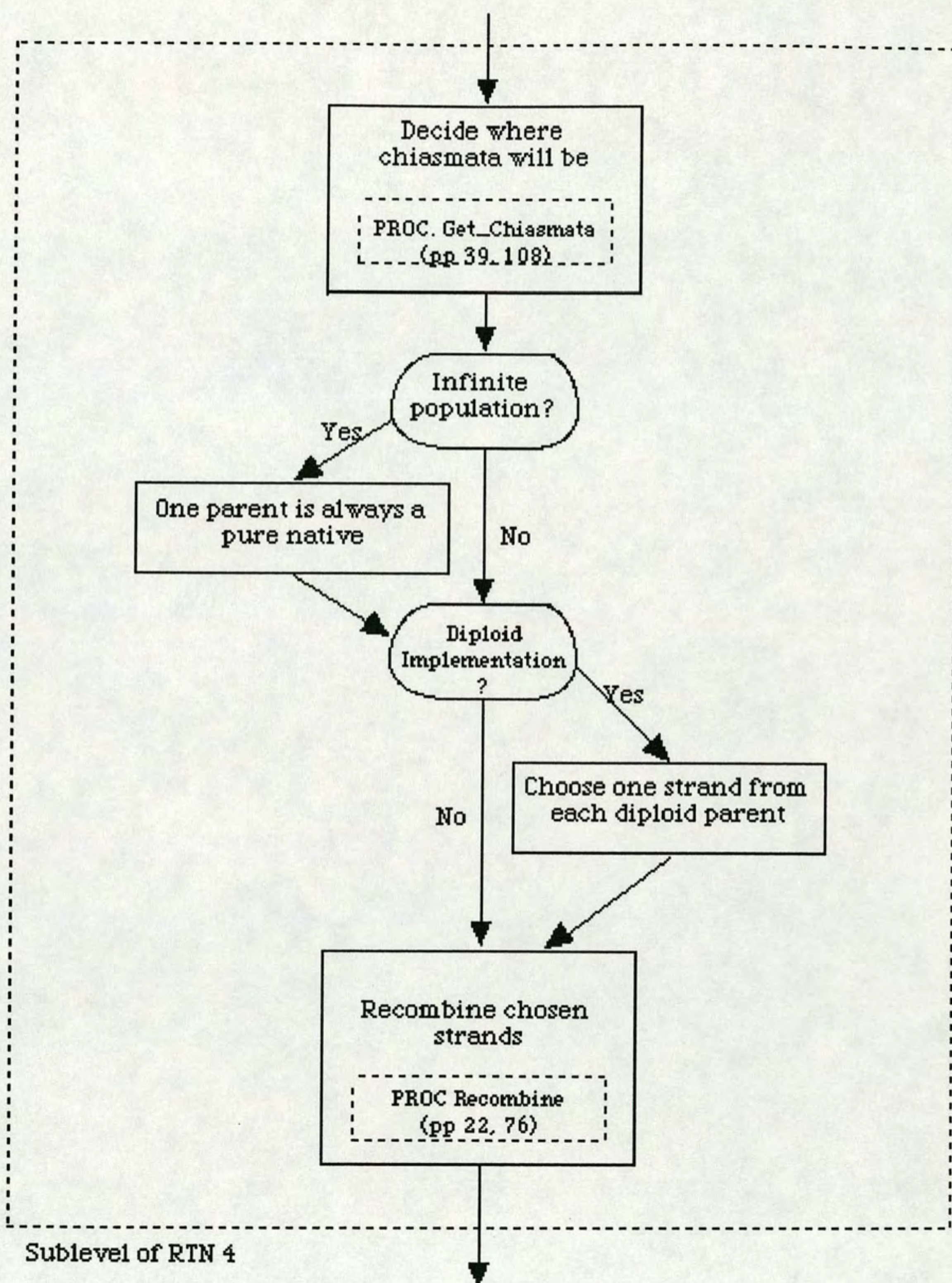

RTN 2: One Replicate (pp62, 151)




RTN 3: Migration (pp64, 148)




RTN 4: Reproduction (pp64, 148)




RTN 5: An Offspring (pp 47, 49, 124, 126)

## A1.1 General unit

```
unit General;
interface
{ TABLE OF DATA TYPE LENGTHS IN BYTES }
 const
{       TYPE    |  LENGTH        }
{---------------------------------}
{}
   length_DOUBLE = 8;         {}
{}
   length_PTR = 4;         {}
{}
   length_REAL = 4;         {}
{}
   length_INTEGER = 2;         {}
{}
   length_LONGINT = 4;         {}
{}
   length_BYTE = 1;         {}
{_______________________________________}
 procedure Longint_to_string (subject: Longint; var s: string);
 procedure real_to_string (subject: real; var s: string);
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure init_chi;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function chisum: real;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure chi (O, E: real);
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function chidf: integer;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function pickAreal: real;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function tossAcoin: boolean;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function uniform: real;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function logit (p, lmax: real): real;
{---------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function poisson (mu: real): integer;
{---------------------------------------------------------------}
implementation


{---------------------------------------------------------}
{ Functions min and max - general purpose                  }
{---------------------------------------------------------}
```

```
function min (a, b: real): real;
begin
  if a < b then
    min := a
  else
    min := b
end;

function max (a, b: real): real;
begin
  if a > b then
    max := a
  else
    max := b
end;


procedure Longint_to_string (subject: Longint; var s: string);
  var
    i: Longint;

begin
  i := abs(subject);
  s := '';
  if subject = 0 then
    s := '0';
  while i > 0 do
    begin
      s := concat(chr(i mod 10 + 48), s);
      i := i div 10;
    end;
  if subject < 0 then
    s := concat('-', s)
end;

procedure real_to_string (subject: real; var s: string);
  var
    i: Longint;
    r: real;
    t: string;

begin
  i := trunc(ABS(subject));
  r := ABS(subject) - i;
  s := '';
  if i = 0 then
    s := '0';
  while i > 0 do
    begin
      s := concat(chr(i mod 10 + 48), s);
      i := i div 10
    end;
  if SUBJECT < 0 then
    S := CONCAT('-', s);
  s := concat(s, '.');
  if r = trunc(r) then
    s := concat(s, '0')
  else
    begin
      r := r * 10;
      repeat
```

```
          t := '';
          t := concat(chr(trunc(r) mod 10 + 48), t);
          s := concat(s, t);
          r := r * 10
        until r = trunc(r)
      end
  end;

  var
    chi_tot: real;
    chi_dfree: integer;
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure init_chi;
{----------------------------------------------------------------}
  begin
    chi_tot := 0;
    chi_dfree := 0;
  end;
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function chisum: real;
{----------------------------------------------------------------}
  begin
    chisum := chi_tot
  end;
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function chidf: integer;
{----------------------------------------------------------------}
  begin
    chidf := chi_dfree;
  end;
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure chi (O, E: real);
{----------------------------------------------------------------}
  begin
    if E >= 5 then
      begin
        chi_tot := (sqr(O - E) / E) + chi_tot;
        chi_dfree := chi_dfree + 1
      end
  end;
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function PickAreal: real;
{----------------------------------------------------------------}
{----------------------------------------------------------------}
    var
      tr: longint;
  begin
    tr := random + bsl(random, 16);
    pickAreal := (0.5 + tr / 4294967296);{works on the Mac}
  end;
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
```

```
function tossAcoin: boolean;
{-----------------------------------------------------------------}
begin
  tossAcoin := random > 0
end;
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
function uniform: real;
{-----------------------------------------------------------------}
  var
    x: real;
begin
  repeat
    x := pickAreal;
  until (x > 0) and (x < 1);
  uniform := x;
end;
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
function logit (p, lmax: real): real;
{-----------------------------------------------------------------}
                    {logit transform}

  var
    l: real;

begin
  if p <= 0 then
    logit := -lmax
  else if p >= 1 then
    logit := lmax
  else
    begin
      l := ln(p / (1 - p));
      if l > lmax then
        logit := lmax
      else if l <= -lmax then
        logit := -lmax
      else
        logit := l
    end
end; {of function logit}
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
function poisson (mu: real): integer;
{-----------------------------------------------------------------}
  var
    i: integer;
    tt, s: real;

begin
  if mu <= 0 then
    poisson := 0
  else
    begin
      i := 0;
      tt := exp(-mu);
      s := uniform;
```

```
      while s > tt do
        begin
          s := s * uniform;
          i := i + 1
        end;
      poisson := i
    end
  end;
{_____________________________________________________________}


end.
```

# A1.2 Junctions unit

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
unit Junctions;
{-----------------------------------------------------------------------}
{ Re-written by Stuart Baird 10/12/91                                              }
{                                              }
{ Two pools of junctions are used, one to hold the existing generation's junctions,   }
{ one to hold the new generation's. When a generation is no longer needed, its pool is }
{ declared empty, and the new and old are swapped to make room for the next gen.    }
{                                              }
{ NB Two properties of this system are important:                                   }
{ 1/ Junctions on a strand are now always in contiguous increasing order in memory }
{ 2/ I'll think of the other later                     }
{______________________________________________________________________________}
interface
  uses
    mymem, general;

  const
    Max_chiasmata = 400;
    nbuckets = 75;

  type
    jindex = LongInt; { for indexing the junction pools }

    strand = record
        start_state: boolean;
        start, last: jindex;    { the index to its start and end in }
      end;                       { the junction array }

    positionArray = array[1..Max_chiasmata] of real;

    chiasmataRec = record
        total: integer;
        positions: positionArray
      end;


    bucketRec = record
        top: real;
        count: LongInt
      end;
  var
    bucket: array[1..nbuckets] of bucketRec;
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure BUBBLE_SORT (var A: positionArray; n: integer);
{-----------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function INSERTION_SORT (var A: positionArray; n: integer): boolean;
{-----------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function INITIALISE_POOLS (heapsize: Longint): Longint;
{-----------------------------------------------------------------------}
{ 11/11/91 This now uses mymem for run time arrays with THINK pascal          }
{ All remaining memory in mymem is split equally between newpool and oldpool      }
{______________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RETURN_POOLS;
{-----------------------------------------------------------------------}
```

```
{ De-allocates memory used by the pool                          }
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SWAP_POOLS (var crash: boolean);
{------------------------------------------------------------------}
{ After recombination Newpool becomes the old pool, and the old pool is no longer      }
{ needed so it is emptied and used as the new newpool                    }
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure INIT_STRAND (var adam: strand; state: boolean);
{------------------------------------------------------------------}
{ Used to initialize homogeneous strands                             }
{__________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RECOMBINE (var crash: boolean; strand1, strand2: strand; var newstrand: strand;
cRec: chiasmataRec);
{------------------------------------------------------------------}
{  Transcription starts on strand1. There are chiasmata_No of crossovers.          }
{  Junctions are drawn from the pool where possible.              }
{__________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function TRUE_STATE (subject: strand): real;
{------------------------------------------------------------------}
{ The number retured is not strictly the heterozygosity, but the fraction of  }
{ the strand that is in the state 'true', as this conveys more information    }
{__________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure emptybuckets;
{------------------------------------------------------------------}
{__________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure setbuckets;
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function bucketTRUE_STATE (subject: strand): real;
{------------------------------------------------------------------}
{ The number retured is not strictly the heterozygosity, but the fraction of}
{ the strand that is in the state 'true', as this conveys more information    }
{__________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function PURE (subject: strand; state: boolean): boolean;
{------------------------------------------------------------------}
{ Is the strand pure state x? }
{__________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function STATE_AT (gposition: real; subject: strand): boolean;
{------------------------------------------------------------------}
{ The state of the strand at the point-gene specified is returned. If there  }
{ is a junction at this point then the state returned is a random boolean.}
{{_________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function USED_RECENTLY: Longint;
{------------------------------------------------------------------}
{ Returns the number of junctions used since the last inquiry, or t=0      }
{__________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure PRINT_CHIASMATA (cRec: chiasmataRec);
{------------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of chiasmata  }
{__________________________________________________________________}
```

```
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure PRINT_STRAND (joe: strand; x: integer);
{------------------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of strand     }
{________________________________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure SAVE_STRAND (joe: strand);
{------------------------------------------------------------------------}
{________________________________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure LOAD_STRAND (var joe: strand);
{------------------------------------------------------------------------}
{________________________________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure REWRITE_SECRET;
{------------------------------------------------------------------------}
{________________________________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure RESET_SECRET;
{------------------------------------------------------------------------}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure CLOSE_SECRET;
{------------------------------------------------------------------------}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure LOAD_NEWPOOL;
{------------------------------------------------------------------------}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure SAVE_NEWPOOL;
{------------------------------------------------------------------------}
{________________________________________________________________________________}

implementation

 const
   infinity = -1;
   text_loci = 69;
   untouched_heap = 0;
 type

   junction = real;


   junctionPool = record
       base, start, size: jindex;
     end;



 var
   oldpool, newpool: junctionPool;
   junctionindex: mymemoryHandle;
   Total_junctions: Longint;
   crash: boolean;
   recent_junctions: Longint;
   secret: file of strand;
   bobbys: file of real;
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure BUBBLE_SORT (var A: positionArray; n: integer);
{------------------------------------------------------------------------}
```

```
    var
      i, j: integer;
      temp: real;

  begin
    for i := 1 to n - 1 do
      for j := n downto i + 1 do
        if A[j] < A[j - 1] then
          begin
            temp := A[j];
            A[j] := A[j - 1];
            A[j - 1] := temp
          end;
  end;{BUBBLE_SORT}
{_______________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function INSERTION_SORT (var A: positionArray; n: integer): boolean;
{--------------------------------------------------------------------}
    var
      j: integer;
      temp: real;
      ok: boolean;

  begin
    ok := true;
    for j := 1 to n - 1 do
      if A[j] = A[n] then
        ok := false;
    if ok then
      for j := n downto 2 do
        if A[j] < A[j - 1] then
          begin
            temp := A[j];
            A[j] := A[j - 1];
            A[j - 1] := temp
          end;
    INSERTION_SORT := ok
  end;{INSERTION_SORT}
{_______________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function INITIALISE_POOLS (heapsize: Longint): Longint;
{--------------------------------------------------------------------}
{ 11/11/91 This now uses mymem for run time arrays with THINK pascal          }
{ All remaining memory in mymem is split equally between newpool and oldpool          }
{_______________________________________________________________________}
    var
      temp: junction;
      ok: boolean;
      i: jindex;
      halfsize: Longint;
  begin
    writeln('Initialising junction pool');
    junctionindex := mmHandle;
    Total_junctions := mmAvail div sizeof(temp);
    halfsize := total_junctions div 2;
    realmem(oldpool.base, halfsize);
```

```
    realmem(newpool.base, halfsize);
    writeln(Total_junctions, ' junctions available.');
    recent_junctions := 0;
    INITIALISE_POOLs := mmAvail - total_junctions * sizeof(temp)
  end;
{_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SWAP_POOLS (var crash: boolean);
{---------------------------------------------------------------------}
{ After recombination Newpool becomes the old pool, and the old pool is no longer }
{ needed so it is emptied and used as the new newpool                             }
{---------------------------------------------------------------------}
    var
      temp_oldpool: JunctionPool;
  begin{bug}
    crash := false;
    temp_oldpool := newpool;

    newpool.base := oldpool.base;
    newpool.start := oldpool.base;
    newpool.size := Total_junctions div 2;

    oldpool := temp_oldpool;
{   recent_junctions := 0}
{bug}
  end;
{_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RETURN_POOLS;
{---------------------------------------------------------------------}
{ De-allocates memory used by the pool                                }
{---------------------------------------------------------------------}
    var
      temp: jindex;
  begin
    if newpool.base < oldpool.base then
      realforget(newpool.base, (Total_junctions div 2) * 2)
    else
      realforget(oldpool.base, (Total_junctions div 2) * 2)
  end;
{_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure INIT_STRAND (var adam: strand; state: boolean);
{---------------------------------------------------------------------}
{ Used to initialize homogeneous strands                              }
{_____________________________________________________________________}
  begin
    adam.start_state := state;
    adam.last := 0;
    adam.start := 1
  end;{INIT_STRAND}
{_____________________________________________________________________}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure RECOMBINE (var crash: boolean; strand1, strand2: strand; var newstrand: strand;
cRec: chiasmataRec);
{--------------------------------------------------------------------}
{ Transcription starts on strand1. There are chiasmata_No of crossovers.        }
{ Junctions are drawn from newpool.                                              }
{____________________________________________________________________}
  label
    999;

  var
    ReadJunctions, IgnoredJunctions, RJlast, IJlast, temp: jindex;
    i: integer;
    read_state, ignored_state, continue: boolean;


 {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure NEW_JUNCTION (var recipient: strand);
 {--------------------------------------------------------------}
 { A pointer to the last junction of a strand is necessary to achieve this}
 { increase in speed. This ony uses order(n) extra storage.        }
 {______________________________________________________________}

  begin
   if not (newpool.size = 0) then
    begin
     if recipient.last = 0 then
      recipient.start := newpool.start;
     recipient.last := newpool.start;
     newpool.start := newpool.start + 1;
     newpool.size := newpool.size - 1;
     recent_junctions := recent_junctions + 1
    end
  end;
 {______________________________________________________________}


begin
 ReadJunctions := strand1.start; { Transcription starts... }
 RJlast := strand1.last;
 read_state := strand1.start_state;  { ...on strand 1.       }
 IgnoredJunctions := strand2.start;
 IJlast := strand2.last;
 ignored_state := strand2.start_state;

 newstrand.start_state := read_state;   { Transcribe start state }
 newstrand.start := newpool.start;        { Connect newstrand to newpool }
 newstrand.last := 0;


 for i := 1 to cRec.total do
  begin{ Chiasmata loop}
   continue := true;

   while (ReadJunctions <= RJlast) and continue do
    begin
     continue := (cRec.positions[i] > junctionindex^^[ReadJunctions]);
```

```
                {chiasmata further on}
          if continue then
            begin { transcribe junction }
              junctionindex^^[newpool.start] := junctionindex^^[ReadJunctions];
              NEW_JUNCTION(newstrand);
              ReadJunctions := ReadJunctions + 1;
              read_state := not (read_state);       { junction=>change of state }
            end
        end;
  { There are now either no more junctions on this strand, or          }
  { ReadJunctions points to the junction after the chiasma.            }
  { Either way we now wish to stop transcribing junctions from this    }
  { strand, and cross over to the ignored strand.                      }
  { NB at this point there may be no more junctions left so be careful}

      continue := true;{ we must determine the state at our current position}
            { on the ignored strand before we cross over          }

      while (IgnoredJunctions <= IJlast) and continue do
        begin
          continue := (cRec.positions[i] >= junctionindex^^[IgnoredJunctions]);
          if continue then
            begin
              ignored_state := not (ignored_state);{ junction=> change of state }
              IgnoredJunctions := IgnoredJunctions + 1
            end;
        end;

      if (read_state <> ignored_state) then{ Chiasma produces..}
        begin
          crash := (newpool.size = 0);
          if crash then
            goto 999;              {..a CRASH  ..or..}
          junctionindex^^[newpool.start] := cRec.positions[i]; {...a junction}
          NEW_JUNCTION(newstrand);
          ignored_state := read_state;
          read_state := not (read_state)       { junction=>change of state  }
        end;
  {we now cross over}
      temp := IgnoredJunctions;
      IgnoredJunctions := ReadJunctions;
      ReadJunctions := temp;
      temp := IJlast;
      IJlast := RJlast;
      RJlast := temp
    end;{chiasmata loop}

  { We now must transcribe any remaining junctions after the last chiasma}
    while (ReadJunctions <= RJlast) do
      begin
        crash := (newpool.size = 0);
        if crash then
          goto 999;
        junctionindex^^[newpool.start] := junctionindex^^[ReadJunctions];
        NEW_JUNCTION(newstrand);
        read_state := not (read_state);       { junction=>change of state  }
        ReadJunctions := ReadJunctions + 1
      end;
999:
  end;{RECOMBINE}
{___________________________________________________________________}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure emptybuckets;
{--------------------------------------------------------------------}
   var
     i: integer;
 begin
   for i := 1 to nbuckets do
     bucket[i].count := 0;
 end;
{____________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure setbuckets;
{--------------------------------------------------------------------}
   var
     i: integer;
 begin
   bucket[1].top := 1.0;
   for i := 2 to nbuckets do
     bucket[i].top := bucket[i - 1].top * 0.8;
 end;
{____________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function TRUE_STATE (subject: strand): real;
{--------------------------------------------------------------------}
{ The number retured is not strictly the heterozygosity, but the fraction of }
{ the strand that is in the state 'true', as this conveys more information    }
{____________________________________________________________________}
   var
     state: boolean;
     j: Longint;
     total, marker: real;

 begin
   marker := 0;
   total := 0;
   state := subject.start_state;
   j := subject.start;

   while (j <= subject.last) do
     begin
       if state then
         total := total + (junctionindex^^[j] - marker)
       else
         marker := junctionindex^^[j];
       state := not (state);
       j := j + 1
     end;
   if state then
     total := total + (1 - marker);
   TRUE_STATE := total
 end;{TRUE_STATE}
{____________________________________________________________________}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function bucketTRUE_STATE (subject: strand): real;
{-----------------------------------------------------------------------}
{ The number retured is not strictly the heterozygosity, but the fraction of  }
{ the strand that is in the state 'true', as this conveys more information     }
{ 8/2/92: When the distribution flag is set the distribution of block sizes is recorded  }
{_______________________________________________________________________}
   var
     state: boolean;
     j: Longint;
     total, marker: real;

  {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
   procedure putinabucket (blocksize: real);
  {----------------------------------------------------------------------}
    var
      i: integer;
   begin
     i := 1;
     while (blocksize <= bucket[i].top) and (i < nbuckets) do
       i := i + 1;
     bucket[i].count := bucket[i].count + 1
   end;
  {_______________________________________________________________________}

 begin
   marker := 0;
   total := 0;
   state := subject.start_state;
   j := subject.start;

   while (j <= subject.last) do
     begin
       if state then
         begin
           total := total + (junctionindex^^[j] - marker);
           putinabucket(junctionindex^^[j] - marker)
         end
       else
         marker := junctionindex^^[j];
       state := not (state);
       j := j + 1
     end;
   if state then
     begin
       total := total + (1 - marker);
       putinabucket(1 - marker)
     end;
   bucketTRUE_STATE := total;
 end;{bucketTRUE_STATE}
{_______________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function PURE (subject: strand; state: boolean): boolean;
{-----------------------------------------------------------------------}
{ Is the strand pure state x? }
{_______________________________________________________________________}
 begin
   pure := (subject.start_state = state) and (subject.last = 0);
 end;{PURE}
```

```
{________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function STATE_AT (gposition: real; subject: strand): boolean;
{-------------------------------------------------------------------}
{ The state of the strand at the point-gene specified is returned. If there  }
{ is a junction at this point then the state returned is a random boolean.}
{________________________________________________________________}
  var
    state, reached: boolean;
    j: Longint;

 begin
   state := subject.start_state;
   j := subject.start;
   reached := false;

   while (j <= subject.last) and not (reached) do
     begin
       reached := (junctionindex^^[j] >= gposition);
       if reached then
         if junctionindex^^[j] = gposition then
           STATE_AT := tossAcoin {approx. p=0.5}
         else
           STATE_AT := state;
       state := not (state);
       j := j + 1
     end;
   if not reached then
     STATE_AT := state
 end;{STATE_AT}
{______________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function USED_RECENTLY: Longint;
{-------------------------------------------------------------------}
{ Returns the number of junctions used since the last inquiry, or t=0      }
{______________________________________________________________}
 begin
   USED_RECENTLY := recent_junctions;
   recent_junctions := 0
 end;
{______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure PRINT_STRAND (joe: strand; x: integer);
{-------------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of strand     }
{________________________________________________________________}
  var
    ReadJunctions: Longint;
    read_state: boolean;
    marker: real;
    i, count, reps: integer;

 begin
```

```
    ReadJunctions := joe.start;
    read_state := joe.start_state;

    write('strand', x : 2, ' ');
    for i := 1 to text_loci do
      write('_');
    writeln;
    write('        |');

    marker := 0;
    count := 0;
    while (ReadJunctions <= joe.last) do
      begin
        reps := round((junctionindex^^[ReadJunctions] - marker) * text_loci);
        if read_state then
          for i := 1 to reps do
            write('@')
        else
          for i := 1 to reps do
            write(' ');
        count := count + reps;
        read_state := not (read_state);        { junction=>change of state }
        marker := junctionindex^^[ReadJunctions];
        ReadJunctions := ReadJunctions + 1
      end;

    if read_state then
      for i := 1 to (text_loci - count) do
        write('@')
    else
      for i := 1 to (text_loci - count) do
        write(' ');

    writeln('|');

    write('%@=', round(TRUE_STATE(joe) * 100) : 3, '   ');
    for i := 1 to text_loci do
      write('"');
    writeln
  end;{PRINT_STRAND}
{_______________________________________________________________________}

{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SAVE_STRAND (joe: strand);
{------------------------------------------------------------------}
{_______________________________________________________________________}
  begin
    with joe do
      begin
        start := start - newpool.base;
        if last = 0 then
          last := -33
        else
          last := last - newpool.base;
      end;
    write(secret, joe);
  end;{SAVE_STRAND}
{_______________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
```

```
procedure LOAD_STRAND (var joe: strand);
{-----------------------------------------------------------------}
{________________________________________________________________}
begin
  read(secret, joe);
  with joe do
    begin
      start := start + newpool.base;
      if last = -33 then
        last := 0
      else
        last := last + newpool.base;
    end;
end;{LOAD_STRAND}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
procedure SAVE_NEWPOOL;
{-----------------------------------------------------------------}
  var
    i: Longint;
{________________________________________________________________}
begin
  rewrite(bobbys, 'h.dls');
  write(bobbys, newpool.start - newpool.base);
  for i := newpool.base to newpool.start - 1 do
    begin
      write(bobbys, junctionindex^^[i]);
    end;
end;{SAVE_NEWPOOL}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
procedure LOAD_NEWPOOL;
{-----------------------------------------------------------------}
  var
    i, tint: Longint;
    temp: real;
{________________________________________________________________}
begin
  reset(bobbys, 'h.dls');
  read(bobbys, temp);
  tint := round(temp);
  newpool.start := newpool.base + tint;
  for i := newpool.base to newpool.start - 1 do
    begin
      read(bobbys, junctionindex^^[i]);
    end;
  close(bobbys);
end;{LOAD_NEWPOOL}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
procedure REWRITE_SECRET;
{-----------------------------------------------------------------}
{________________________________________________________________}
begin
  rewrite(secret, 'h.oz');
```

```
  end;{REWRITE_SECRET}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RESET_SECRET;
{----------------------------------------------------------------}
{________________________________________________________________}
  begin
    reset(secret, 'h.oz');
  end;{RESET_SECRET}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure CLOSE_SECRET;
{----------------------------------------------------------------}
{________________________________________________________________}
  begin
    CLOSE(secret);
  end;{CLOSE_SECRET}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure PRINT_CHIASMATA (cRec: chiasmataRec);
{----------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of chiasmata  }
{________________________________________________________________}
    var
      i, j: integer;
      marker: real;

  begin
    marker := 0;
    write('Chiasma: ');
    for i := 1 to (cRec.total) do
      begin
        for j := 1 to (round((cRec.positions[i] - marker) * text_loci) - 1) do
          write(' ');
        write('|');
        marker := cRec.positions[i]
      end;
    writeln
  end;{PRINT_CHIASMATA}
{________________________________________________________________}

end.
```

## A1.3 Haploid unit

```
{SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS}
{____________________________________________}
{| ST PROSPERO PASCAL => MAC THINK PASCAL CONVERSION:
|( Stuart Baird 24/10/91 )}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS}

unit Haploid;

interface

  uses
    Junctions;


  const
    infinitedememax = 60000;
    infinity = -1;
    maxgenes = 100;
    maxdemes = 100;
    maxinds = 10000;
    dummy0 = 0;
    dummy1 = 1;
    checkseln = false;
    checkmigrn = false;
    logitmax = 5;{bounds on the logit function}

  type
    bittype = BOOLEAN;
    haplotype = STRAND;
    genetype = -1..maxgenes;
    demetype = 1..maxdemes;
    indtype = LongInt;

    demearray = array[1..infinitedememax] of haplotype;
    demearrayptr = ^demearray;

    poptype = array[1..dummy1] of demearrayptr;
    popptrtype = ^poptype;

    samplearray = array[0..dummy0] of real;
    samplearrayptr = ^samplearray;

    dtdemetype = array[0..dummy0] of samplearrayptr;
    dtdemeptrtype = ^dtdemetype;

    dttype = array[0..dummy0] of real;
    dtptrtype = ^dttype;

    smallrealpvectype = array[0..dummy0] of real; { for haploids }
    smallrealpvecptrtype = ^smallrealpvectype;

    outarraytype = array[1..dummy1, 1..dummy1] of real;
    outarrayptrtype = ^outarraytype;

    posarray = array[1..20] of real;
```

```
{-------------------------------------------------------------------}
  procedure puthap (var p: popptrtype; h: haplotype; d: demetype; i: indtype);
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure savehap (var p: popptrtype; d: demetype; i: indtype);
{-------------------------------------------------------------------}
  procedure loadhap (var p: popptrtype; d: demetype; i: indtype);
{-------------------------------------------------------------------}
  procedure swappop (var p, np: popptrtype);
{-------------------------------------------------------------------}
  function search (var t: smallrealpvecptrtype; x: real; nn: Longint): Longint;
{-------------------------------------------------------------------}
  procedure swapind (var a, b: indtype);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure transfer (var pop: popptrtype; var newpop: popptrtype; mfromleft, mfromright,
mtoleft, mtoright: longint; demeno: demetype; ninds: Longint);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure migrate (var pop, newpop: popptrtype; nmig, nmigbarr, ndemes: integer; ninds:
LongInt; nmidleft, nmidright: integer);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure immigrate (var pop: popptrtype; nmigbarr, ndemes: integer; allONE: strand);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure double_immigrate (var pop: popptrtype; nmig, nmigbarr, ndemes: integer; allONE,
allZero: strand);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_immigrate (var pop: popptrtype; nmig, nmigbarr, ndemes: integer; allONE,
allZero: strand; var ninds: longint);
{-------------------------------------------------------------------}

implementation

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure puthap (var p: popptrtype; h: haplotype; d: demetype; i: indtype);
{-------------------------------------------------------------------}
  begin
    p^[d]^[i] := h
  end;
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure savehap (var p: popptrtype; d: demetype; i: indtype);
{-------------------------------------------------------------------}
  begin
    save_strand(p^[d]^[i]);
  end;
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure loadhap (var p: popptrtype; d: demetype; i: indtype);
{-------------------------------------------------------------------}
  begin
    load_strand(p^[d]^[i]);
```

```
  end;
{____________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure swappop (var p: popptrtype; var np: popptrtype);
{------------------------------------------------------------}
    var
      tptr: popptrtype;
  begin
    tptr := p;
    p := np;
    np := tptr;
  end;
{____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function search (var t: smallrealpvecptrtype; x: real; nn: Longint): Longint;
{------------------------------------------------------------}
{Looks for the position of x in a table of reals;t^[0]=0,t^[j]-t^[j-1]=w[j].}
{0<x<t^[imax]; returns j if t^[j-1]<x<=t^[j]}
{____________________________________________________________}
    var
      i, imax, imin: Longint;
  begin
    if nn <= 1 then
      search := 1
    else
      begin
        imax := nn;
        imin := 0;{set the initial interval; x must lie between}
{                                 t^[imax] and  t^[imin]}
        repeat
          i := (imax + imin) div 2;
          if x > t^[i] then
            imin := i
          else
            imax := i
        until imax = imin + 1;
        if x = t^[imin] then
          search := imin
        else
          search := imax
      end
  end;
{____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure swapind (var a, b: indtype);
{------------------------------------------------------------}
    var
      z: indtype;
  begin
    z := a;
    a := b;
    b := z
  end;
{____________________________________________________________}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure transfer (var pop: popptrtype; var newpop: popptrtype; mfromleft, mfromright,
mtoleft, mtoright: longint; demeno: demetype; ninds: Longint);
{--------------------------------------------------------------------}
    var
      leftdeme, thisdeme, rightdeme: demearrayptr;
      offset: integer;
      i: indtype;
      h: haplotype;
  begin
    if mfromleft > 0 then
      leftdeme := pop^[demeno - 1];
    thisdeme := pop^[demeno];
    if mfromright > 0 then
      rightdeme := pop^[demeno + 1];

    if mfromleft + mfromright <> mtoleft + mtoright then
      writeln('ERROR in transfer: asymmetric migration');
    if mfromleft > 0 then
      for i := 1 to mfromleft do
        puthap(newpop, leftdeme^[i], demeno, i);
    if mfromright > 0 then
      for i := ninds downto ninds - mfromright + 1 do
        puthap(newpop, rightdeme^[i], demeno, i);
    offset := mtoright - mfromleft;
    for i := mfromleft + 1 to ninds - mfromright do
      puthap(newpop, thisdeme^[i], demeno, i);
  end;
{____________________________________________________________________}



{""""""""[sublevel of RTN 3 p8 ]""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure migrate (var pop, newpop: popptrtype; nmig, nmigbarr, ndemes: integer; ninds:
LongInt; nmidleft, nmidright: integer);
{____________________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    k := round(nmig / 2);
    kb := round(nmigbarr / 2);
    if ndemes = 2 then
      begin
        transfer(pop, newpop, 0, kb, 0, kb, 1, ninds);
        transfer(pop, newpop, kb, 0, kb, 0, 2, ninds);
      end;
    if ndemes = 3 then
      begin
        transfer(pop, newpop, 0, kb, 0, kb, 1, ninds);
        transfer(pop, newpop, kb, k, kb, k, 2, ninds);
        transfer(pop, newpop, k, 0, k, 0, 3, ninds);
      end;
    if ndemes > 3 then
      begin
        transfer(pop, newpop, 0, k, 0, k, 1, ninds);
        if nmidleft - 1 > 1 then
          for d := 2 to nmidleft - 1 do
            transfer(pop, newpop, k, k, k, k, d, ninds);
```

```
        if nmidright + 1 < ndemes then
          for d := nmidright + 1 to ndemes - 1 do
            transfer(pop, newpop, k, k, k, k, d, ninds);
        transfer(pop, newpop, k, 0, k, 0, ndemes, ninds);
        transfer(pop, newpop, kb, k, kb, k, nmidright, ninds);
        transfer(pop, newpop, k, kb, k, kb, nmidleft, ninds)
      end;
  end;
{______________________________________________________________}


{""""""""[sublevel of RTN 3 p8 ]""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure immigrate (var pop: popptrtype; nmigbarr, ndemes: integer; allONE: strand);
{---------------------------------------------------------------------}
{______________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    for d := 1 to ndemes do
      for i := 1 to nmigbarr do
        puthap(pop, allONE, d, i); { Pure invaders! }
{ NB no need to swap pops!}
  end;
{______________________________________________________________}


{""""""""[sublevel of RTN 3 p8 ]""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure double_immigrate (var pop: popptrtype; nmig, nmigbarr, ndemes: integer; allONE,
allZero: strand);
{---------------------------------------------------------------------}
{ added 9/7/92: prevents fixation in a finite population  }
{______________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    for d := 1 to ndemes do
      begin
        for i := 1 to nmigbarr do
          puthap(pop, allONE, d, i); { Pure invaders! }
        for i := nmigbarr + 1 to nmig + nmigbarr + 1 do
          puthap(pop, allZERO, d, i); { Pure natives! }
      end;{ NB no need to swap pops!}
  end;
{______________________________________________________________}


{""""""""[sublevel of RTN 3 p8 ]""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_immigrate (var pop: popptrtype; nmig, nmigbarr, ndemes: integer; allONE,
allZero: strand; var ninds: longint);
{---------------------------------------------------------------------}
{______________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    for d := 1 to ndemes do
      for i := ninds + 1 to nmigbarr + ninds + 1 do
        puthap(pop, allONE, d, i); { Pure invaders! }
```

```
    ninds := ninds + nmigbarr;
{ NB no need to swap pops!}
  end;
{____________________________________________________________}

end.
```

## A1.4 Stats unit

```
unit stats;

interface
  uses
    General, Junctions, Haploid;
{graph;}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure INITIALISE_POSITIONS (nloci, nmarkergenes: integer);
{--------------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure GET_CHIASMATA (nloci, nmarkergenes: integer; mu: real; var cRec:
chiasmataRec);
{---------------------------------------------------------------------------}
{______________________________________________________________________________}

{
  }
  function fitness (s, x, B: real; sfunction: integer): real;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - }
{____________________________________________________________________________}
{
  }
  function getgenefreq (ninds: indtype; var d: demearrayptr; g: genetype): real;
{--------------------------------------------------------------------------}
{ frequency of a gene in a particular deme                }
{____________________________________________________________________________}
    {vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv}
  procedure find_wbar (s, B: real; sfunction: integer; ninds: indtype; nmarkergenes: genetype; pp:
popptrtype; d: demetype; var wb, varw, hb, varh, ESone, ESzero: real; var onetot, zerotot: integer);
{--------------------------------------------------------------------------}
{ of deme                                  }
{__________________________________________________________________}
    {vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv}
  procedure bucketfind_wbar (s, B: real; sfunction: integer; ninds: indtype; nmarkergenes:
genetype; pp: popptrtype; d: demetype; var wb, varw, hb, varh, ESone, ESzero: real; var onetot,
zerotot: integer);
{--------------------------------------------------------------------------}
 { see suzuki page 813                          }
 {________________________________________________________________}
{
  }
  procedure genefreqstats (ninds: indtype; nmarkergenes: genetype; var pp: popptrtype; d:
demetype; var genic_variance: real; var pbar: real);
{--------------------------------------------------------------------------}
{________________________________________________________________}
{
  }
  function total_variance (ninds: indtype; nmarkergenes: genetype; var pp: popptrtype; d:
demetype): real;
{--------------------------------------------------------------------------}
{________________________________________________________________}


implementation
```

```
var
  positions: array[1..maxgenes] of real;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure INITIALISE_POSITIONS (nloci, nmarkergenes: integer);
{-----------------------------------------------------------------------}
    var
      i: integer;

  begin
    if nmarkergenes = 1 then
      positions[1] := 0.5;
    for i := 1 to nmarkergenes do
      positions[i] := (i - 1) / (nmarkergenes - 1);
    if nloci <> infinity then
      for i := 1 to nmarkergenes do
        positions[i] := (round(positions[i] * (nloci - 1)) + 0.5) / nloci;
  end;
{______________________________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure GET_CHIASMATA (nloci, nmarkergenes: integer; mu: real; var cRec:
chiasmataRec);
{----------[ sublevel of RTN 5, p10 ]----------------------------------------}
{______________________________________________________________________________}
    var
      i, x, pair, nmpairs, regionpairs, nchiasmata2, chtot: integer;
      region: real;
      c: char;
  begin
    nmpairs := nmarkergenes - 1;
    chtot := 0;
    nchiasmata2 := Poisson(mu);{ Number of chiasmata on TWO chromatids }


    if nloci = infinity then
      begin
        for i := 1 to nchiasmata2 do
          cRec.positions[chtot + i] := pickAreal; {/ nmpairs) + region}
        chtot := chtot + nchiasmata2;
      end

    else if nchiasmata2 > 0 then{ finite number of loci }
      begin
        writeln('You havent fixed this');
        regionpairs := round(nloci * (positions[pair + 1] - positions[pair]));
        x := trunc(uniform * regionpairs);
        cRec.positions[chtot + 1] := ((x + 0.5) / nloci) + positions[pair];
        i := 2;
        while i <= nchiasmata2 do
          begin
            regionpairs := round(nloci * (positions[pair + 1] -
                                              positions[pair]));
            x := trunc(uniform * regionpairs);
            cRec.positions[chtot + i] := ((x + 0.5) / nloci) + positions[pair];
            if INSERTION_SORT(cRec.positions, chtot + i) then
              i := i + 1;
```

```
        end;
      chtot := chtot + nchiasmata2;
    end;

   if nloci = infinity then
     BUBBLE_SORT(cRec.positions, chtot);
   cRec.total := chtot;
 end;{GET_CHIASMATA}
{__________________________________________________________________}



{
  }
 function fitness (s, x, B: real; sfunction: integer): real;
{-------[sublevel of RTN 4, p9 ]-----------------------------------------}
{_______________________________________________________________}
 begin
   case sfunction of
     1:
       fitness := 1 - (s * x);            { additive: s<=f<=1 }
     2:
       fitness := 1 - exp(x * ln(s));            { 1-(s^x) : multiplicative: s<=f<=1 }
     3:
       fitness := 1 - s * (4 * x * (1 - x));   { epistatic: s<=f<=1 }
     4:
       fitness := 1 - s * exp(B * ln(4 * x * (1 - x)));
                                          { epistatic: Glaciated valley }
   end;
 end;
{_______________________________________________________________}



{
  }
 function getgenefreq (ninds: indtype; var d: demearrayptr; g: genetype): real;
{------------------------------------------------------------------}
{ frequency of a gene in a particular deme                    }
{_______________________________________________________________}
   var
     sum: real;
     i: indtype;

 begin
   sum := 0;
   for i := 1 to ninds do
     if STATE_AT(positions[g], d^[i]) then
       sum := sum + 1;
   getgenefreq := sum / ninds
 end;{getgenefreq}
{_______________________________________________________________}

   {^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^}
 procedure find_effective_selection (joe: haplotype; nmarkergenes: genetype; var zerocount,
onecount: integer);
{------------------------------------------------------------------}
{ Added  12:40pm 22/1/92  Stuart
}
{_____________________________________________________________}
   var
```

```
    g: integer;
  begin
    zerocount := 0;
    onecount := 0;
    for g := 1 to nmarkergenes do
      if STATE_AT(positions[g], joe) then
        onecount := onecount + 1
      else
        zerocount := zerocount + 1
  end;

    {~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~}
  procedure find_wbar (s, B: real; sfunction: integer; ninds: indtype; nmarkergenes: genetype; pp:
popptrtype; d: demetype; var wb, varw, hb, varh, ESone, ESzero: real; var onetot, zerotot: integer);
 {--------------------------------------------------------------------------}
 { see suzuki page 813                                          }
 {________________________________________________________________________}
    var
      i: indtype;
      h, smh, ssqh, w, smw, ssqw: real;
      zerocount, onecount: integer;
  begin
    smh := 0;
    ssqh := 0;
    smw := 0;
    ssqw := 0;
    onetot := 0;
    zerotot := 0;
    ESone := 0;
    ESzero := 0;
    for i := 1 to ninds do
      begin
        h := TRUE_STATE(pp^[d]^[i]);
        w := fitness(s, h, B, sfunction);
        smh := smh + h;
        ssqh := ssqh + sqr(h);
        smw := smw + w;
        ssqw := ssqw + sqr(w);
        find_effective_selection(pp^[d]^[i], nmarkergenes, zerocount,
            onecount);
        ESone := ESone + (onecount * w);
        ESzero := ESzero + (zerocount * w);
        onetot := onetot + onecount;
        zerotot := zerotot + zerocount;
      end;
    hb := smh / ninds;
    varh := (ssqh / ninds) - sqr(hb);
    wb := smw / ninds;
    varw := (ssqw / ninds) - sqr(wb);
    ESone := Esone / onetot;
    Eszero := ESzero / zerotot;
  end;
 {________________________________________________________________________}

    {~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~}
  procedure bucketfind_wbar (s, B: real; sfunction: integer; ninds: indtype; nmarkergenes:
genetype; pp: popptrtype; d: demetype; var wb, varw, hb, varh, ESone, ESzero: real; var onetot,
zerotot: integer);
 {--------------------------------------------------------------------------}
 { see suzuki page 813                                          }
 {________________________________________________________________________}
```

```
    var
      i: indtype;
      h, smh, ssqh, w, smw, ssqw: real;
      zerocount, onecount: integer;
  begin
    smh := 0;
    ssqh := 0;
    smw := 0;
    ssqw := 0;
    onetot := 0;
    zerotot := 0;
    ESone := 0;
    ESzero := 0;
    for i := 1 to ninds do
      begin
        h := bucketTRUE_STATE(pp^[d]^[i]);
        w := fitness(s, h, B, sfunction);
        smh := smh + h;
        ssqh := ssqh + sqr(h);
        smw := smw + w;
        ssqw := ssqw + sqr(w);
        find_effective_selection(pp^[d]^[i], nmarkergenes, zerocount,
            onecount);
        ESone := ESone + (onecount * w);
        ESzero := ESzero + (zerocount * w);
        onetot := onetot + onecount;
        zerotot := zerotot + zerocount;
      end;
    hb := smh / ninds;
    varh := (ssqh / ninds) - sqr(hb);
    wb := smw / ninds;
    varw := (ssqw / ninds) - sqr(wb);
    ESone := Esone / onetot;
    Eszero := ESzero / zerotot;
  end;
 {_____________________________________________________________________}


    {^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^}
  procedure genefreqstats (ninds: indtype; nmarkergenes: genetype; var pp: popptrtype; d:
demetype; var genic_variance: real; var pbar: real);
{-----------------------------------------------------------------------}
{_____________________________________________________________________}
    var
      g: genetype;
      p, GVsum, Psum: real;
  begin
    GVsum := 0;
    Psum := 0;
    for g := 1 to nmarkergenes do
      begin
        p := getgenefreq(ninds, pp^[d], g);
        GVsum := GVsum + p * (1 - p);    { assuming haploidy }
        Psum := Psum + p;
      end;
    genic_variance := GVsum;
    pbar := Psum / nmarkergenes;
  end;
{_____________________________________________________________________}
```

```
{
  }
  function total_variance (ninds: indtype; nmarkergenes: genetype; var pp: popptrtype; d:
demetype): real;
{----------------------------------------------------------------------}
{________________________________________________________________________}
    var
      k: indtype;
      g: genetype;
      p, sum, sum2, zind: real;
  begin
    sum := 0;
    sum2 := 0;
    for k := 1 to ninds do
      begin
        zind := 0;
        for g := 1 to nmarkergenes do
          if STATE_AT(positions[g], pp^[d]^[k]) then
            zind := zind + 1;
        sum := sum + zind;
        sum2 := sum2 + sqr(zind);
      end;
    total_variance := (sum2 / ninds) - sqr(sum / ninds);
                              { assuming haploidy }
  end;
{________________________________________________________________________}


end.
```

# A1.5 Haploid Junctions program

```
program haploid_junctions;
{ Created February/March 1991 by Stuart Baird,        }
{ using much code from the program:                   }
{ Multiloc  which was:                                 }
{ created June 1990: modified version of multloc9;    }
{ modified by Nick, September 1990                    }
{ modified by Nick, March     1991                    }
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{ 11/11/91 This now uses mymem for run-time arrays with THINK pascal }
{----------------------------------------------------------------}
{ 22/1/92 This now outputs Effective selection pressures }
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{ units of routines                                               }
{----------------------------------------------------------------}
{ 7/7/92 This will now simulate a single infinite deme          }
{----------------------------------------------------------------}
 uses
{graphics;}
   mymem, junctions, Haploid, general, stats;
{________________________________________________________________}
 const
{   harddisc = 'Hard disc';}
    harddisc = 'Quadra HD';
{   harddisc = 'Macintosh HD';}
 label
   999, 333;

 var
   t, dt, sample, nsamples: integer;
   nmig, nmigbarr: Longint;
   ndemes, nmidleft, nmidright: longint;
   nmarkergenes: genetype;
   nloci: genetype;
   ninds: LongInt;
   smallcumwptr: smallrealpvecptrtype;

 var
   array_heap, pool_heap: Longint;
   run: string;
   out_file, out_JperDeme, out_Pbar, out_Gvar, out_ESP, out_DIST: Text;
   out_Tvar, out_Vdiseq, out_Wbar, out_Wvar, bf, out_hvar, out_hbar, context: Text;
   in_file: file of integer;
   out_fname, batch_fname, path: string;
   out_jname, out_pbname, out_gvname, out_hvname, out_ESPname, out_distname: string;
   out_tvname, out_vdqname, out_wbname, out_wvname, out_hbname: string;
   control_string, TimeString, DateString: string;
   batch, selfrandom, done, crash, explode, immigration, infinitedeme: boolean;
   ans, ans2, ans3: string;
   pop, newpop: popptrtype;
   s, B, rec, mu, mig, migbarr, hwidth: real;
   replicates, repNo, twarm, tmax, runint: integer;
   seedNo, ninit, nfix, g: integer;
   demeno: demetype;
   pbar, pvar, dbar: dtdemeptrtype;
   allOne, allZero: haplotype; {'pure' haplotypes}
   ip, iq, i1, i2, i3, i4, h: haplotype;
   d: demetype;
   i: indtype;
```

```
    demearraylength, poparraylength, pvecarraylength: Longint;

    windowrect: rect;
{ for opening folders 7/8/92 }
    error: OSerr;
    name: stringptr;
    creator, tipe: OStype;
    vol, unknownint: integer;
    unknownLongint, savedninds: Longint;
    replicatesummary: string;
    sfunction, completeruns, repstart, pausedthisrun, tstart: integer;
    precalculate: real;
    distribution, paused, breakoff: boolean;
    newpoint, oldpoint: point;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure select (pop: popptrtype; d: demetype);
{--------------[sublevel of RTN 4, p9 ]------------------------------------}
{ sets up a table of cumulative fitnesses for a deme> This is a crafty trick}
{ from multilocus - the subsequent search of this table will return an entry}
{ in proportion to the real "interval" it occupies.                    }
{________________________________________________________________________}
    var
      i: indtype;

  begin
    smallcumwptr^[0] := 0;
    for i := 1 to ninds do
      smallcumwptr^[i] := smallcumwptr^[i - 1] + fitness(s, TRUE_STATE(pop^[d]^[i]), B,
sfunction);
  end;
{________________________________________________________________________}




{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure parents (pop: popptrtype; d: demetype; var mum, dad: haplotype);
{--------------------------------------------------------------------}
{ returns individuals with probability proportional to their fitness, using }
{ the table of cumulative fitness set in by select                     }
{________________________________________________________________________}
    var
      i1, i2: indtype;

  begin
    i1 := search(smallcumwptr, pickAreal * smallcumwptr^[ninds], ninds);
    i2 := search(smallcumwptr, pickAreal * smallcumwptr^[ninds], ninds);
     {cumwptr^[i-1]<rand<=cumwptr^[i]}
    if tossAcoin then
      swapind(i1, i2);
    mum := pop^[d]^[i1];
    dad := pop^[d]^[i2];
  end;
{________________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure reproduce (var pop: popptrtype; var newpop: popptrtype; rec: real);
```

```
{------------[sublevel of RTN 4, p9 ]----------------------------------}
   label
     999;

   var
     d, chNo, sink: integer;
     jim: indtype;
     mum, dad, sonnyjim: haplotype;
     chiasmata: chiasmataRec;
     c: char;
     totr, time: real;
 begin
   time := t;
   for d := 1 to ndemes do
     begin
{set up the table of cumulative fitnesses for this deme}
       select(pop, d);
       for jim := 1 to ninds do { [ RTN 5, p10 ] }
         begin
{set up recomb. record, a sorted array of nchiasmata positions }
           GET_CHIASMATA(nloci, nmarkergenes, mu, chiasmata);
{draw parents, using the table of cumulative fitnesses set up by SELECT}
           parents(pop, d, mum, dad);
{recombination produces haploid baby}
           RECOMBINE(crash, mum, dad, sonnyjim, chiasmata);
           if crash then
             goto 999;
{put the new individual into newpop}
           puthap(newpop, sonnyjim, d, jim);
         end;
{record the number of junctions in the new deme}
       if ((t + 1) >= twarm) and ((t + 1) mod dt = 0) then
         begin
           if d = 1 then
             write(out_JperDeme, time + 1 : 3 : 1, '    ');
           write(out_JperDeme, USED_RECENTLY / ninds : 10 : 8); {junctions per ind for deme}
           if d = ndemes then
             writeln(out_JperDeme)
           else
             write(out_JperDeme, '    ')
         end
       else
         sink := USED_RECENTLY;
     end;
999:
 end;
{_________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure parent (pop: popptrtype; d: demetype; var mumordad: haplotype);
{ 29/6/92 infinite population variant }
{----------------------------------------------------------------}
{ returns individuals with probability proportional to their fitness, using }
{ the table of cumulative fitness set in by select                        }
{_________________________________________________________________}
   var
     i1: indtype;

 begin
```

```
    i1 := search(smallcumwptr, pickAreal * smallcumwptr^[ninds], ninds);
     {cumwptr^[i-1]<rand<=cumwptr^[i]}
    mumordad := pop^[d]^[i1];
  end;
{_____________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_reproduce (var pop: popptrtype; var newpop: popptrtype; rec: real);
{ 29/6/92 infinite population variant }
{-----------[sublevel of RTN 4, p9 ]------------------------------------}
    label
      999;

    var
      d, chNo, sink: integer;
      jim: indtype;
      mumordad, sonnyjim: haplotype;
      chiasmata: chiasmataRec;
      c: char;
      totr, time, wtot: real;
      offspring, x, y, z: integer;
      new_ninds: longint;
  begin
    x := 0;
    y := 0;
    z := 0;
    wtot := 0;
    new_ninds := 0;
    time := t;
    for d := 1 to ndemes do
      begin
{ don't set up the table of cumulative fitnesses for this deme}
{   select(pop, d);}
        for jim := 1 to ninds do  { [ RTN 5, p10 ] }
          begin

{ SIGNIFICANTLY different from finite n case: }
{ infinite population => wbar==1 therefore fitness is nolonger relative to other individuals, }
{ but to 1. Cumulative table is therefore unnecessary }
{ Yet another major departure: Selection removes blocks length Y at a given rate in Nicks model, }
{ all individuals are now parents who produce Poisson(fitness) offspring in the next generation }

            for offspring := 1 to Poisson(2 * fitness(s, TRUE_STATE(pop^[d]^[jim]), B, sfunction))
do
              begin
                GET_CHIASMATA(nloci, nmarkergenes, mu, chiasmata);
{   if chiasmata.total = 0 then}
{   y := y + 1;}
{set up recomb. record, a sorted array of nchiasmata positions }
{ DON'T draw ONE parent, using the table of cumulative fitnesses set up by SELECT}
{   parent(pop, d, mumordad); }
{ALWAYS recombination between PURE NATIVE ( zero ) and recombinant/immigrant produces
haploid baby}
                if tossAcoin then
                  RECOMBINE(crash, allZERO, pop^[d]^[jim], sonnyjim, chiasmata)
                else
                  RECOMBINE(crash, pop^[d]^[jim], allZERO, sonnyjim, chiasmata);

                if crash then
                  goto 999;
```

```
{put the new individual into newpop IF IT IS A RECOMBINANT}
                if not PURE(sonnyjim, false) then { false = ZERO = native }
                  begin
                    new_ninds := new_ninds + 1;
                    if new_ninds >= infinitedememax then
                      begin
                        explode := true;
                        goto 999;
                      end;
                    puthap(newpop, sonnyjim, d, new_ninds);
                  end
{    else}
{  sonnyjim should be pure! }
{    begin}
{    wtot := wtot + true_state(sonnyjim);}
{    x := x + 1;}
{    if chiasmata.total = 0 then}
{    z := z + 1;}
{    end;}
              end;
          end;
{record the new population size}
        ninds := new_ninds;
{    writeln('pure native offspring:  ', x);}
{    writeln('average H . I .: ', wtot / x);}
{    writeln('immigrants/recos: ', ninds);}
{    writeln('pairings with no chmta: ', y);}
{    writeln('.. which gave pure natives: ', z);}

{record the number of junctions in the new deme}
        if ((t + 1) >= twarm) and ((t + 1) mod dt = 0) then
          begin
            if d = 1 then
              write(out_JperDeme, time + 1 : 3 : 1, '    ');
            write(out_JperDeme, USED_RECENTLY / ninds : 10 : 8, '   ', ninds : 10); {junctions per
ind for deme, ninds}
            if d = ndemes then
              writeln(out_JperDeme)
            else
              write(out_JperDeme, '   ')
          end
        else
          sink := USED_RECENTLY;
      end;
999:
  end;
{_______________________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure askstuff;
{------------------------------------------------------------------}
{_______________________________________________________________________________}
    var
      gg: genetype;
  begin

    writeln('The program runs several replicates; each is run for tmax generations.');
    writeln('Statistics are recorded every dt gens, after an initial delay of twarm gens');
    writeln('Summarise this replicate in not more than 27 characters');
    writeln('---------------------------');
```

```
    readln(replicatesummary);
    writeln('tmax, dt, twarm ? ');
    replicates := 1;
    readln(tmax, dt, twarm);

    writeln('Haploid individuals migrate.For each haploid in the next generation,');
    writeln('two parent haploids are chosen with probability proportional to their fitness.');
    writeln('The haploid is generated by recombination from the two parents.');
    writeln('# of demes (<=', maxdemes : 3, '); # of haploid inds (<=', maxinds : 3, ', -1 approximates
infinity)');
    readln(ndemes, ninds);
    writeln;
    writeln('# of loci to be simulated (-1 approximates infinity), # of marker genes');
    readln(nloci, nmarkergenes);
    writeln('The genes are on a chromosome, total map length R.');
    writeln('R?');
    readln(rec);
    writeln('Selection:');
    writeln('  Form of selection ,1=additive (1-sp), 2=multiplicative (1- s^x), ');
    writeln('  3=epistatic (1 - s  (4*x (1 - x))), 4=glaciated');
    write('   ');
    readln(sfunction);
    if sfunction = 4 then {glaciated}
      begin
        writeln('  beta?');
        readln(B);
      end;
    write('   s?');
    readln(s);
    writeln('nmig individuals migrate (half in each direction); at the centre,');
    writeln('there is a barrier,');
    writeln('across which nmigbarr migrate. If there is no barrier, set nmig=nmigbarr');
    writeln('In the case of a single deme, nmigbarr is the number of foreign immigrants');
    writeln('nmig the number of native immigrants');
    writeln('nmig (a multiple of 2),nmigbarr (a multiple of 2) ');
    readln(nmig, nmigbarr);

    writeln('seed for random numbers ? ');
    readln(seedNo);

    mig := nmig / ninds;

  end;
{_______________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure askstuff_batch;
{---------------------------------------------------------------}
{_______________________________________________________________________}
    var
      gg: genetype;
  begin
    readln(bf);
    readln(bf, replicatesummary);
    readln(bf);
    readln(bf, replicates, tmax, dt, twarm);
    readln(bf);
    readln(bf, ndemes, ninds, nloci, nmarkergenes);
```

```
    readln(bf);
    readln(bf, rec, sfunction);
    if sfunction = 4 then {glaciated}
      begin
        readln(bf);
        readln(bf, B)
      end
    else
      B := 1;
    readln(bf, s);
    readln(bf);
    readln(bf, nmig, nmigbarr);

    readln(bf);
    readln(bf, ans);
    selfrandom := (ans = 'y');

    mig := nmig / ninds;
  end;
{____________________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure set_listfile;
{------------------------------------------------------------------------}
  begin
    writeln(out_file, 'Simulating an infinite locus cline');
    writeln;
    writeln(out_file, 'Run ', run, ' time ', TimeString, ' date ', DateString);
    writeln;
    writeln(out_file, 'runInt, Replicates, repNo, nsamples,twarm,tmax,dt,seedNo:');
    writeln(out_file, runInt);
    writeln(out_file, replicates);
    writeln(out_file, repNo);
    writeln(out_file, nsamples);
    writeln(out_file, twarm);
    writeln(out_file, tmax);
    writeln(out_file, dt);
    writeln(out_file, seedNo);
    writeln(out_file, 'ndemes,ninds,nloci,nmarkergenes:');
    writeln(out_file, ndemes);
    writeln(out_file, ninds);
    writeln(out_file, nloci);
    writeln(out_file, nmarkergenes);
    writeln(out_file, 'sfunction,Beta,s,rec,nmig,nmigbarr,mig:');
    writeln(out_file, sfunction, ' ', B);
    writeln(out_file, s : 5 : 4);
    writeln(out_file, rec : 5 : 4);
    writeln(out_file, nmig);
    writeln(out_file, nmigbarr);
    writeln(out_file, mig);
    writeln(out_file);
  end;
{____________________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure init_dist_file;
{------------------------------------------------------------------------}
```

```
    var
      i, j: integer;
{--------------------------------------------------------------}
  begin
    write(out_DIST, 0.0 : 3 : 1, '   ');{t=0 also}
    for j := 1 to ndemes do
      for i := 1 to nbuckets do
        write(out_DIST, bucket[i].top : 10 : 8, '  '); {interval of distribution}
    writeln(out_DIST);
  end;
{--------------------------------------------------------------}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure initialise;
{--------------------------------------------------------------}
  begin
    nmidleft := (ndemes div 2);
    nmidright := (nmidleft + 1);
    RandSeed := seedNo; {SSSSSSSSSSSSSS}
    t := tstart;
  end;
{______________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure initrep;
{--------------------------------------------------------------}
    var
      i: longint;
      d, demeno: integer;
  begin

    if immigration then
      begin
        if nmigbarr = 0 then { 1000 immigrant ( One ) individuals in an infinite native ( zero ) pop
}
          begin
            ninds := 1000;
            for d := 1 to ndemes do
              for i := 1 to ninds do
                puthap(newpop, allOne, d, i){all genes set to one}
          end
        else
          for d := 1 to ndemes do
            for i := 1 to ninds do
              puthap(newpop, allZero, d, i){all genes set to zero}
      end
    else
      begin
        if nmig = 0 then
          for d := 1 to ndemes do
            begin
              for i := 1 to ninds do
                puthap(newpop, allZero, d, i);
              for i := 1 to ninds div 2 do
                puthap(newpop, allOne, d, i);{half genes set to 1}
            end
        else
          begin
```

```
          for d := 1 to nmidleft do
            for i := 1 to ninds do
              puthap(newpop, allZero, d, i); {all genes set to zero}
          for d := nmidright to ndemes do
            for i := 1 to ninds do
              puthap(newpop, allOne, d, i); {all genes set to one}
        end;
      end;
  end;
{_______________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_initrep;
{-----------------------------------------------------------------}
  begin
    ninds := 0; { all individuals are pure natives, and therefore can be disregarded }
    if paused then
      ninds := savedninds;
  end;
{_______________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure update_stats;
{-----------------------------------------------------------------}
{_______________________________________________________________}
{updates the list of statistics: zbar,Vgenic,Vdiseq,wbar for selected loci,}
{collected every dt generations}
    var
      d: demetype;
      g: genetype;
      wb, vw, hb, vh, tv, ESone, ESzero: real;
      gv, pbar, time: real;
      onetot, zerotot, i: integer;
  begin
    time := t;
    distribution := ((t * dt * 4) mod tmax = 0) and (t <> 0);
    write(out_Hbar, time : 3 : 1, '  ');
    write(out_Hvar, time : 3 : 1, '  ');
    write(out_Pbar, time : 3 : 1, '  ');
    write(out_Gvar, time : 3 : 1, '  ');
    write(out_Tvar, time : 3 : 1, '  ');
    write(out_Vdiseq, time : 3 : 1, '  ');
    write(out_Wbar, time : 3 : 1, ' ');
    write(out_Wvar, time : 3 : 1, ' ');
    write(out_ESP, time : 3 : 1, ' ');
    for d := 1 to ndemes do
      begin
        if distribution then
          bucketfind_wbar(s, B, sfunction, ninds, nmarkergenes, pop, d, wb, vw, hb, vh, ESone,
ESzero, onetot, zerotot)
        else
          find_wbar(s, B, sfunction, ninds, nmarkergenes, pop, d, wb, vw, hb, vh, ESone, ESzero,
onetot, zerotot);
        genefreqstats(ninds, nmarkergenes, pop, d, gv, pbar);
        tv := total_variance(ninds, nmarkergenes, pop, d);

        if infinitedeme then
          begin
            if t = 0 then
```

```
        hb := 1;
      write(out_Hbar, HB : 10 : 8, '   ', precalculate / (HB * ninds) : 10 : 8, ' ');
    end
  else
    write(out_Hbar, HB : 10 : 8, '   ');
  write(out_Hvar, vh : 10 : 8, '    ');
  write(out_Pbar, pbar : 10 : 8, ' ');
  write(out_Gvar, gv : 10 : 8, '    ');
  write(out_Tvar, tv : 10 : 8, ' ');
  write(out_Vdiseq, tv - gv : 10 : 8, '    ');  { Vdiseq_sel }
  write(out_Wbar, wb : 10 : 8, ' ');
  write(out_Wvar, vw : 10 : 8, ' ');
  write(out_ESP, ESone : 10 : 8, '    ', ESzero : 10 : 8, '    ', onetot : 10, '    ', zerotot : 10, '
');
  if distribution then
    begin
      rewrite(out_Dist, concat(out_distname, ' t=', stringof(time)));
      for i := 1 to nbuckets do
        writeln(out_DIST, bucket[i].top : 10 : 8, '  ', bucket[i].count : 10);
      close(out_DIST);
      emptybuckets;
      distribution := false;
    end;
  end;
 writeln(out_Pbar);
 writeln(out_Hvar);
 writeln(out_Hbar);
 writeln(out_Gvar);
 writeln(out_Tvar);
 writeln(out_Vdiseq);  { Vdiseq_sel }
 writeln(out_Wbar);
 writeln(out_Wvar);
 writeln(out_ESP);
{ Latest addition **********************}
{   if (HB = 1) and not infinitedeme then}
{   t := tmax - 1;}
{*******************************}
 end;
{________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure get_seed;  { sets new random seed }
{--------------------------------------------------------------}
  var
    d: dateTimeRec;
 begin
  getTime(d);
  seedNo := (d.second * d.minute) + d.day
 end;
{________________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure get_time;  { sets up time and date strings }
{--------------------------------------------------------------}
  var
    d: dateTimeRec;
 begin
  getTime(d);
```

```
    DateString := stringof(d.day : 2, '/', d.month : 2, '/', d.year : 2);
    TimeString := stringof(d.hour : 2, ':', d.minute : 2, ':', d.second : 2);
  end;
{_______________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure get_run;   { sets up run string }
{---------------------------------------------------------------------}
  begin
    reset(in_file, 'HP.inf');
    read(in_file, runInt);
    if paused then
      runInt := runInt - 1;
    Longint_to_string(runInt, run);
    rewrite(in_file);
    write(in_file, runInt + 1);
    close(in_file)
  end;
{_______________________________________________________________________}



{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure INITIALISE_ARRAYS;
{---------------------------------------------------------------------}
    var
      top: Longint;
      ok: boolean;
{---------------------------------------------------------------------}
    procedure init_poparray (var p: popptrtype);
      var
        d: integer;
    begin

      newmem(p, poparraylength, ok);
      if not ok then
        writeln('Uh oh, population not initialised');
      for d := 1 to ndemes do
        begin
          newmem(p^[d], demearraylength, ok);
          if not ok then
            writeln('Uh oh, demearray not initialised')
        end;
    end;
{---------------------------------------------------------------------}

  begin
    top := mmAvail;
    demearraylength := ninds * sizeof(allOne);
      {array      :=[1..ninds] of haplotype      }

    poparraylength := ndemes * length_PTR;
     {array      :=[1..ndemes] of demearrayptr}

    pvecarraylength := (ninds + 2) * length_REAL;
      {array      :=[0..maxinds+1] of        REAL}


    init_poparray(pop);
    init_poparray(newpop);
```

```
    newmem(smallcumwptr, pvecarraylength, ok);
    if not ok then
      writeln('Uh oh, cumulative table not initialised');
    array_heap := top - mmAvail;
    writeln('Array heap ', array_heap);
  end;
{_______________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RETURN_ARRAYS;
{-----------------------------------------------------------------}
    procedure ret_poparray (var p: popptrtype);
      var
        d: integer;
    begin
      for d := 0 to ndemes - 1 do
        dispmem(p^[ndemes - d], demearraylength);
      dispmem(p, poparraylength);
    end;
{-----------------------------------------------------------------}
  begin

    dispmem(smallcumwptr, pvecarraylength);
    ret_poparray(newpop);
    ret_poparray(pop);
  end;
{_______________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure init_jperdeme_file;
{-----------------------------------------------------------------}
    var
      i: integer;
{-----------------------------------------------------------------}
  begin
    write(out_Jperdeme, 0.0 : 3 : 1, '    ');{t=0 also}
    if infinitedeme then
      for i := 1 to ndemes + 1 do
        write(out_JperDeme, 0.0 : 10 : 8, '    ') {initially 0 JPI, ninds=0 }
    else
      for i := 1 to ndemes do
        write(out_JperDeme, 0.0 : 10 : 8, '    '); {initially 0 JPI }
    writeln(out_JperDeme);
  end;
{-----------------------------------------------------------------}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure get_context;
{-----------------------------------------------------------------}
    var
      d, i, tally: integer;
      state: string;
{-----------------------------------------------------------------}
  begin
    reset(context, 'h.con');
```

```
    readln(context, state);
    paused := state <> 'finished';
    if paused then
      begin
        tally := 0;
        readln(context, batch_fname);
        readln(context, completeruns);
        readln(context, pausedthisrun);
        readln(context, savedninds);
        readln(context, tStart);
        if state = 'processing' then
{ state=processing: machine has crashed, or rude interruption. }
        {Population data will be corrupted, therefore }
          begin
            tStart := 0;
            Savedninds := 0;
            pausedthisrun := 0;
          end;
        close(context);
        rewrite(context, 'h.con');
        writeln(context, 'processing');
        writeln(context, batch_fname);
        writeln(context, completeruns);
        writeln(context, pausedthisrun);
        writeln(context, savedninds);
        writeln(context, tStart);

        reset(bf, batch_fname);
        repeat
          readln(bf, control_string);
          askstuff_batch;
          tally := tally + replicates
        until tally > completeruns;
        repStart := replicates - (tally - completeruns - 1);
        t := tstart;
      end;
    close(context);
  end;
{-----------------------------------------------------------------}

{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RETURN_AFTER_BREAK;
{-----------------------------------------------------------------}
    var
      i, j: longint;
      d: integer;
{-----------------------------------------------------------------}
  begin
    RESET_SECRET;
    for d := 1 to ndemes do
      for i := 1 to savedninds do
        begin
          loadhap(newpop, d, i);
        end;
    CLOSE_SECRET;
    LOAD_NEWPOOL;
  end;
{-----------------------------------------------------------------}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
```

```
procedure save_context;
{-----------------------------------------------------------------}
  var
    d, tally: integer;
    i, j: longint;
{-----------------------------------------------------------------}
begin
  for i := 1 to 20 do
    writeln('SAVING DATA IN PROGRESS, PLEASE WAIT..');
  REWRITE_SECRET;
  for d := 1 to ndemes do
    for i := 1 to ninds do
      begin
        savehap(newpop, d, i);
      end;
  SAVE_NEWPOOL;
  rewrite(context, 'h.con');
  writeln(context, 'cleanly paused'); { this state can only be reached at this point }
  writeln(context, batch_fname);
  writeln(context, completeruns);
  writeln(context, pausedthisrun + 1);
  writeln(context, ninds);
  writeln(context, t);
  close(context);
  for i := 1 to 20 do
    writeln;
  writeln('Done, thankyou.');
end;
{-----------------------------------------------------------------}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
procedure update_context;
{-----------------------------------------------------------------}
begin
  rewrite(context, 'h.con');
  writeln(context, 'processing');
  writeln(context, batch_fname);
  writeln(context, completeruns);
  writeln(context, pausedthisrun);
  writeln(context, savedninds);
  writeln(context, tStart);
  close(context);
end;
{-----------------------------------------------------------------}

{-----------------------------------------------------------------}
procedure check_for_breaks;
{-----------------------------------------------------------------}
  var
    temp: boolean;
begin
  getmouse(newpoint);

  temp := breakoff;
  if not breakoff then
    breakoff := button
  else if button then
    breakoff := false;
  if temp <> breakoff then
    writeln('Breaking = ', not breakoff);
```

```
    if breakoff then
      oldpoint := newpoint;

    if ((newpoint.v > oldpoint.v + 3) or (newpoint.v < oldpoint.v - 3)) or ((newpoint.h > oldpoint.h
+ 3) or (newpoint.h < oldpoint.h - 3)) then
      begin
        save_context;
        goto 333;
      end;

    oldpoint := newpoint;
    if crash or explode then
      goto 999;
  end;
{---------------------------------------------------------------------}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{"""""""""""""""[ RTN 1 p 6 ]"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
begin
  completeruns := 0;
  pausedthisrun := 0;
  repstart := 1;
  tstart := 0;
  breakoff := false;
  get_context;

  windowrect.topleft.v := 50;
  windowrect.topleft.h := 0;
  windowrect.botright.v := 380;
  windowrect.botright.h := 510;
  SetTextRect(windowRect);
  ShowText;
  installmem(paused, done);
  if not done then
    goto 333;
  INIT_STRAND(allOne, true);
  INIT_STRAND(allZero, false);
  write('Do you want to read parameters from a batch file ? ');
  if paused then
    begin
      batch := true;
      writeln('Auto reading....');
    end
  else
    begin
      readln(ans);
      batch := (ans = 'y');
      if batch then
        begin
          batch_fname := OldFileName('Select Batch File');
          reset(bf, batch_fname);
          readln(bf, control_string)
        end
      else
        control_string := 'go_ahead';
    end;
  new(name);
  name^ := harddisc;
```

```
  error := getvol(name, vol);
  creator := 'SJEB';
  tipe := 'Jntn';


{-----------------------------------------------------------------}
{       Batch loop                                                  }
{-----------------------------------------------------------------}
  while control_string = 'go_ahead' do
    begin
      if not paused then
        begin
          if batch then
            askstuff_batch
          else
            askstuff;
        end;
      immigration := ndemes = 1;
      infinitedeme := ninds = -1;
      if infinitedeme then
        begin
          ninds := infinitedememax; { for array initialisation }
          precalculate := nmigbarr * (1 - s) / s; { for computing s* }
        end;
      nsamples := ((tmax - twarm) div dt) + 1;
     {rec defines a Poisson distribution of recombination events }
{    mu := (-1 * ln(1 - 2 * rec)) * 0.5; }
     { Haldane's map function }
     {halved as we are only considering two of four chromatids    }
{ changed as of 7/8/92. We now input the total map length}
      mu := rec;
      writeln('Available memory is ', mmAvail : 10);
      INITIALISE_ARRAYS;
      INITIALISE_POSITIONS(nloci, nmarkergenes);{positions of marker genes}
      writeln('Available memory is ', mmAvail : 10);

      pool_heap := INITIALISE_POOLS(mmAvail);
      writeln('Available memory is ', mmAvail : 10);
      setbuckets;
      getmouse(oldpoint);

{-----------------------------------------------------------------}
{      Replicate loop      [ RTN 2 p7 ]                             }
{-----------------------------------------------------------------}

      path := concat(harddisc, ':Desktop Folder:Stuart:Stuarts Results:', replicatesummary);
      error := dircreate(vol, unknownint, path, unknownLongint);
      for repNo := repstart to replicates do
        begin
          explode := false;
          get_run;
          get_time;
          writeln('Run ', run, '. Replicate', repNo, ' of', replicates);
          if batch then
            begin
              if selfrandom then
                get_seed
              else
                readln(bf, seedNo);
              writeln('Random seed: ', seedNo)
            end;
```

```
            path := concat(harddisc, ':Desktop Folder:Stuart:Stuarts Results:', replicatesummary);
            if paused then
              path := concat(path, ':R', run, stringof(pausedthisrun))
            else
              path := concat(path, ':R', run);
            error := dircreate(vol, unknownint, path, unknownLongint);
            path := concat(path, ':R', run);
            out_fname := concat(path, ' Info ');
            out_jname := concat(path, ' JperDeme');
            out_pbname := concat(path, '  Pbar');
            out_hvname := concat(path, ' Hvar');
            out_hbname := concat(path, ' Hbar');
            out_vdqname := concat(path, ' Vdiseq');
            out_gvname := concat(path, ' Gvar');
            out_tvname := concat(path, ' Tvar');
            out_wbname := concat(path, ' Wbar');
            out_wvname := concat(path, ' Wvar');
            out_ESPname := concat(path, ' ESP');
            out_distname := concat(path, ' DIST');
            rewrite(out_file, out_fname);
            rewrite(out_JperDeme, out_jname);
            rewrite(out_Pbar, out_pbname);
            rewrite(out_Hvar, out_hvname);
            rewrite(out_Hbar, out_hbname);
            rewrite(out_Gvar, out_gvname);
            rewrite(out_Tvar, out_tvname);
            rewrite(out_Vdiseq, out_vdqname);
            rewrite(out_Wbar, out_wbname);
            rewrite(out_Wvar, out_wvname);
            rewrite(out_ESP, out_ESPname);
            initialise;
            nfix := 0;
            set_listfile;

            SWAP_POOLS(crash);
            if twarm = 0 then
              init_jperdeme_file;
{   init_dist_file;}
            if infinitedeme then
              infinite_initrep
            else
              initrep;
            tstart := 0;

            if paused then
              RETURN_AFTER_BREAK;
            paused := false;
            swappop(pop, newpop);

{----------------------------------------------------------------}
{       Generation loop                                          }
{----------------------------------------------------------------}
            repeat
              if (t >= twarm) and (t mod dt = 0) then
                begin
                  sample := sample + 1;
                  write('Updating stats for generation = ', t : 2, ' ');
                  update_stats;
                  get_time;
                  writeln('Time ', timeString, ' ', dateString);
                end;
```

```
{as the old population is now unnecessary, we can reclaim all its storage}
{I haven't bothered creating "dummies": I just have no migration beyond the ends}
{[ RTN 3 p8]}
          if not immigration and (nmig > 0) then
            begin
              migrate(pop, newpop, nmig, nmigbarr, ndemes, ninds, nmidleft, nmidright);
              swappop(pop, newpop);
            end
          else if immigration and (nmigbarr > 0) then
            begin
              if infinitedeme then
                infinite_immigrate(pop, nmig, nmigbarr, ndemes, allONE, allZero, ninds)
              else if nmig = 0 then
                immigrate(pop, nmigbarr, ndemes, allONE)
              else
                double_immigrate(pop, nmig, nmigbarr, ndemes, allONE, allZero)
            end;
{[ RTN 4 p9]}
          if infinitedeme then
            infinite_reproduce(pop, newpop, rec)
          else
            reproduce(pop, newpop, rec);
          t := t + 1;

          CHECK_FOR_BREAKS;

          SWAP_POOLS(crash);
          swappop(pop, newpop);
        until t >= tmax;
{-------------------------------------------------------------}
{       End of generation loop                                 }
{-------------------------------------------------------------}

        if (t >= twarm) and (t mod dt = 0) then
          begin
            sample := sample + 1;
            write('Updating stats for generation = ', t : 2, ' ');
            update_stats;
            get_time;
            writeln('Time ', timeString, ' ', dateString);
          end;
999:
        if crash then
          begin
            writeln(out_file, 'Out of Memory.. run terminated on run', run);
            writeln('OUT OF MEMORY... This run is terminated.')
          end;
        if explode then
          begin
            writeln(out_file, 'Population explosion.. run terminated on run', run);
            writeln('POPULATION EXPLOSION... This run is terminated.')
          end;
        close(out_file);
        close(out_JperDeme);
        close(out_Pbar);
        close(out_Hvar);
        close(out_Hbar);
        close(out_Gvar);
        close(out_Tvar);
        close(out_Vdiseq);
        close(out_Wbar);
```

```
        close(out_Wvar);
        close(out_ESP);
        pausedthisrun := 0;
        completeruns := completeruns + 1;
        UPDATE_CONTEXT;
      end;
{----------------------------------------------------------------}
{       End of Replicate loop                                    }
{----------------------------------------------------------------}
    repstart := 1;
    RETURN_POOLS;
    RETURN_ARRAYS;
    writeln('Memory returned, ', mmAvail, ' bytes available.');
    if batch then
      readln(bf, control_string)
    else
      control_string := 'finished';
  end;
{----------------------------------------------------------------}
{       End of Batch loop                                        }
{----------------------------------------------------------------}
333:
end.
```

# Appendix 2
# Inplementation of Diploid Populations with Junctions between N states

# A2.0 Junctions unit

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
unit Junctions;
{   MULTIPLE STATES AND DIPLOID INDIVDUALS CATERED FOR }
{-----------------------------------------------------------------------}
{ Re-written by Stuart Baird 10/12/91                                          }
{                                       }
{ Two pools of junctions are used, one to hold the existing generation's junctions,    }
{ one to hold the new generation's. When a generation is no longer needed, its pool is  }
{ declared empty, and the new and old are swapped to make room for the next gen.    }
{                                       }
{ NB Two properties of this system are important:                               }
{ 1/ Junctions on a strand are now always in contiguous increasing order in memory }
{ 2/ I'll think of the other later                        }
{Re-written by Stuart Baird 24/9/93}
{...to allow 1/mixing of npossstates types of material instead of just two}
{ The position of a junction is stored in the fractional portion of each real number}
{ the state that it changes the strand to is stored in the integer part.}
{NB junctions never take the value 1 or 0, changes are signified as:}
{?}
{to allow ancestor tracing  by unique strand id's, changes are signified as:}
{+}
{changed 24/11/93 to cope with diploids}
{______________________________________________________________}
interface
  uses
    mymem, general, binary;

  const
    Max_chiasmata = 400;
    nbuckets = 75;
    contributers = 1000;

  type
    jindex = LongInt;  { for indexing the junction pools }
    state = integer;{?}

    strand = record
        SID: LongInt;           {(+) strand indetification number, for anc. tracing}
        MSID, FSID: LongInt; {(+) ID's of mother and father }
        start_state: state;{?}
        start, last: jindex;  { the index to its start and end in }
      end;                       { the junction array }

    positionArray = array[1..Max_chiasmata] of real;

    chiasmataRec = record
        total: integer;
        positions: positionArray
      end;


    bucketRec = record
        top: real;
```

```
      count: LongInt
    end;

   bucketrack = array[1..nbuckets] of bucketRec;
   contrack = array[1..contributers] of real;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure BUBBLE_SORT (var A: positionArray;
               n: integer);
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function INSERTION_SORT (var A: positionArray;
               n: integer): boolean;
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function INITIALISE_POOLS (heapsize: Longint): Longint;
{------------------------------------------------------------------}
{ 11/11/91 This now uses mymem for run time arrays with THINK pascal              }
{ All remaining memory in mymem is split equally between newpool and oldpool       }
{__________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure RETURN_POOLS;
{------------------------------------------------------------------}
{ De-allocates memory used by the pool                             }
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure SWAP_POOLS (var crash: boolean);
{------------------------------------------------------------------}
{ After recombination Newpool becomes the old pool, and the old pool is no longer      }
{ needed so it is emptied and used as the new newpool                        }
{------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure INIT_STRAND (var adam: strand;
               start_state: state);{?}
{+}
{------------------------------------------------------------------}
{ Used to initialize homogeneous strands                                     }
{__________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure force_junction (var recipient: strand;
               position: real);
{------------------------------------------------------------------}
{ to produce strange starting individuals }
{__________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure RECOMBINE (var crash: boolean;
               strand1, strand2: strand;
               var newstrand: strand;
               cRec: chiasmataRec);
{------------------------------------------------------------------}
{  Transcription starts on strand1. There are chiasmata_No of crossovers.          }
{  Junctions are drawn from the pool where possible.                   }
{__________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function TRUE_STATE (haplo: strand;
               qstate: state;
               nchromosomes: integer): real;{?}
{------------------------------------------------------------------}
{{ The number retured is not strictly the heterozygosity, but the fraction of  }
{ the strand that is in the state 'true', as this conveys more information    }
{__________________________________________________________________________}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function BINARY_ONECOUNT (subject: strand;
                nbases: LongInt): integer;
{----------------------------------------------------------------------}
{ The number returned is the number of 1s in the binary pool corresponding to }
{ the sections of original genome which make up the strand }
{______________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure emptybuckets (var bucket, conts: bucketrack);
{----------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure zeroConts (var contributions: contrack);
{----------------------------------------------------------------------}
{______________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure setbuckets (var bucket, conts: bucketrack;
                n: integer);
{----------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function bucketCOUNT_STATE (var contributions: contrack;
                var bucket: bucketrack;
                subject: strand;
                justzeroes: boolean): integer;
{----------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function PURE (subject: strand;
                qstate: state): boolean;{?}
{----------------------------------------------------------------------}
{ Is the strand pure state x? }
{______________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function DOMINANCE (haplotype1, haplotype2: strand;
                nchromosomes: integer): real;
{----------------------------------------------------------------------}
{ compares the two strands and calculates the proportion which has at least one 1 allele}
{______________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function HETEROZYGOSITY (haplotype1, haplotype2: strand;
                nchromosomes: integer;
                first_chr: boolean): real;
{----------------------------------------------------------------------}
{ compares the two strands and calculates the proportion which is heterozygous}
{______________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function bucketHETEROZYGOSITY (var bucket: bucketrack;
                haplotype1, haplotype2: strand;
                nchromosomes: integer): real;
{----------------------------------------------------------------------}
{ compares the two strands and calculates the proportion which is heterozygous}
{ dumps block sizes at the same time somehow!!}
{______________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function STATE_AT (gposition: real;
                subject: strand): state;{?}
{----------------------------------------------------------------------}
{ The state of the strand at the point-gene specified is returned. If there   }
{ is a junction at this point then the state returned is minus 1.}
{?}
{?}
{?}
```

```
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function USED_RECENTLY: Longint;
{---------------------------------------------------------------}
{ Returns the number of junctions used since the last inquiry, or t=0      }
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure PRINT_CHIASMATA (cRec: chiasmataRec);
{---------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of chiasmata  }
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure PRINT_STRAND (joe: strand;
               x: integer);
{---------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of strand     }
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SAVE_STRAND (joe: strand);
{---------------------------------------------------------------}
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure LOAD_STRAND (var joe: strand);
{---------------------------------------------------------------}
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure REWRITE_SECRET (destination: string);
{---------------------------------------------------------------}
{_______________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure  RESET_SECRET;
{---------------------------------------------------------------}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure  CLOSE_SECRET;
{---------------------------------------------------------------}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure LOAD_NEWPOOL;
{---------------------------------------------------------------}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SAVE_NEWPOOL (destination: string);
{---------------------------------------------------------------}
{_______________________________________________________________}
```

```
implementation

  const
    infinity = -1;
    text_loci = 69;
    untouched_heap = 0;
  type

    junction = real;{ a tuple of position and state changed to }


    junctionPool = record
        base, start, size: jindex;
      end;



  var
    uniqueSID: LongInt; {used to trace ancestry}
    oldpool, newpool: junctionPool;
    junctionindex: mymemoryHandle;
    Total_junctions: Longint;
    crash: boolean;
    recent_junctions: Longint;
    secret: file of strand;
    bobbys: file of real;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure BUBBLE_SORT (var A: positionArray;
                 n: integer);
{----------------------------------------------------------------}
   var
     i, j: integer;
     temp: real;

 begin
   for i := 1 to n - 1 do
     for j := n downto i + 1 do
       if A[j] < A[j - 1] then
         begin
           temp := A[j];
           A[j] := A[j - 1];
           A[j - 1] := temp
         end;
 end;{BUBBLE_SORT}
{________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function INSERTION_SORT (var A: positionArray;
                 n: integer): boolean;
{----------------------------------------------------------------}
   var
     j: integer;
     temp: real;
     ok: boolean;

 begin
   ok := true;
   for j := 1 to n - 1 do
```

```
      if A[j] = A[n] then
        ok := false;
    if ok then
      for j := n downto 2 do
        if A[j] < A[j - 1] then
          begin
            temp := A[j];
            A[j] := A[j - 1];
            A[j - 1] := temp
          end;
    INSERTION_SORT := ok
  end;{INSERTION_SORT}
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function INITIALISE_POOLS (heapsize: Longint): Longint;
{---------------------------------------------------------------------}
{ 11/11/91 This now uses mymem for run time arrays with THINK pascal              }
{ All remaining memory in mymem is split equally between newpool and oldpool       }
{_______________________________________________________________}
    var
      temp: junction;
      ok: boolean;
      i: jindex;
      halfsize: Longint;
  begin
    UniqueSID := 1;
    writeln('UniqueSID counter zeroed.');
    writeln('Initialising junction pool');
    junctionindex := mmHandle;
    Total_junctions := mmAvail div sizeof(temp);
    halfsize := total_junctions div 2;
    realmem(oldpool.base, halfsize);
    realmem(newpool.base, halfsize);
    writeln(Total_junctions, ' junctions available.');
    recent_junctions := 0;
    INITIALISE_POOLs := mmAvail - total_junctions * sizeof(temp)
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SWAP_POOLS (var crash: boolean);
{---------------------------------------------------------------------}
{ After recombination Newpool becomes the old pool, and the old pool is no longer      }
{ needed so it is emptied and used as the new newpool                         }
{---------------------------------------------------------------------}
    var
      temp_oldpool: JunctionPool;
  begin{bug}
    crash := false;
    temp_oldpool := newpool;

    newpool.base := oldpool.base;
    newpool.start := oldpool.base;
    newpool.size := Total_junctions div 2;

    oldpool := temp_oldpool;
```

```
{   recent_junctions := 0}
{bug}
 end;
{_____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure RETURN_POOLS;
{---------------------------------------------------------------}
{ De-allocates memory used by the pool                          }
{---------------------------------------------------------------}
   var
     temp: jindex;
 begin
   if newpool.base < oldpool.base then
     realforget(newpool.base, (Total_junctions div 2) * 2)
   else
     realforget(oldpool.base, (Total_junctions div 2) * 2)
 end;
{_____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function TUPLE (ajpos: real;
                 astate: state): junction;{?}
{---------------------------------------------------------------}
{ makes a tuple}
{_____________________________________________________________}
 begin
   tuple := ajpos + astate;
 end;{TUPLE}
{_____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function NEXTSTATE (ajunc: junction): state;{?}
{---------------------------------------------------------------}
{ pulls the nextstate from a tuple}
{_____________________________________________________________}
 begin
   nextstate := trunc(ajunc);
 end;{NEXTSTATE}
{_____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function JPOS (ajunc: real): real;
{---------------------------------------------------------------}
{ pulls the junction position from a tuple}
{_____________________________________________________________}
 begin
   jpos := ajunc - trunc(ajunc);
 end;{JPOS}
{_____________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure INIT_STRAND (var adam: strand;
                 start_state: state);{?}
```

```
{----------------------------------------------------------------------}
{ Used to initialize homogeneous strands                               }
{______________________________________________________________________}
 begin
   adam.SID := uniqueSID;
   uniqueSID := uniqueSID + 1;
   adam.FSID := -1;
   adam.MSID := -1;
   adam.start_state := start_state;{?}
   adam.last := 0;
   adam.start := 1
 end;{INIT_STRAND}
{______________________________________________________________________}


 {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure NEW_JUNCTION (var recipient: strand);
 {--------------------------------------------------------------}
 { A pointer to the last junction of a strand is necessary to achieve this}
 { increase in speed. This ony uses order(n) extra storage.          }
 {_____________________________________________________________________}

 begin
   if not (newpool.size = 0) then
     begin
       if recipient.last = 0 then
         recipient.start := newpool.start;
       recipient.last := newpool.start;
       newpool.start := newpool.start + 1;
       newpool.size := newpool.size - 1;
       recent_junctions := recent_junctions + 1
     end
 end;
 {_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure force_junction (var recipient: strand;
                  position: real);
{----------------------------------------------------------------}
{ to produce strange starting individuals }
{______________________________________________________________________}
 begin
   junctionindex^^[newpool.start] := position;
   NEW_JUNCTION(recipient);
 end;
{_____________________________________________________________________}



{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure RECOMBINE (var crash: boolean;
                  strand1, strand2: strand;
                  var newstrand: strand;
                  cRec: chiasmataRec);
{----------------------------------------------------------------}
{ Transcription starts on strand1. There are chiasmata_No of crossovers.       }
{ Junctions are drawn from newpool.                                             }
{______________________________________________________________________}
   label
```

```
    999;

  var
    ReadJunctions, IgnoredJunctions, RJlast, IJlast, temp: jindex;
    i: integer;
    read_state, ignored_state, temp_state: state;{?}
    continue: boolean;
    temp_strand: strand;
    cstart: integer;


begin
  if cRec.total <> 0 then
    if cRec.positions[1] = 0 then
      begin
        temp_strand := strand1;
        strand1 := strand2;
        strand2 := temp_strand;
        cstart := 2
      end
    else
      cstart := 1;
  ReadJunctions := strand1.start;  { Transcription starts... }
  RJlast := strand1.last;
  read_state := strand1.start_state;  { ...on strand 1.         }
  IgnoredJunctions := strand2.start;
  IJlast := strand2.last;
  ignored_state := strand2.start_state;
  if RJlast - ReadJunctions > 10 then
    begin { stops in here }
    end;
  if IJlast - IgnoredJunctions > 10 then
    begin { stops in here }
    end;
  newstrand.SID := uniqueSID;                {ID for tracing }
  UniqueSID := uniqueSID + 1;
  newstrand.MSID := strand1.SID;             {Mother}
  newstrand.FSID := strand2.SID;              {Father}
  newstrand.start_state := read_state;    { Transcribe start state }
  newstrand.start := newpool.start;          { Connect newstrand to newpool }
  newstrand.last := 0;


  for i := cstart to cRec.total do
    begin{ Chiasmata loop}
      continue := true;
{ first we copy the selected strand until we reach the chiasmata point in question}
      while (ReadJunctions <= RJlast) and continue do
        begin
          continue := (cRec.positions[i] > jpos(junctionindex^^[ReadJunctions]));{?}
                {chiasmata further on}
          if continue then
            begin { copy junction }
              junctionindex^^[newpool.start] := junctionindex^^[ReadJunctions];
              NEW_JUNCTION(newstrand);
              read_state := nextstate(junctionindex^^[ReadJunctions]); { junction=>change of state }
              ReadJunctions := ReadJunctions + 1;
            end
        end;
```

```
{ There are now either no more junctions on this strand, or        }
{ ReadJunctions points to the junction AT OR AFTER the chiasma, and           }
{ Read_state indicates the state COMMING UP TO that junction. }
{ Either way we now wish to stop transcribing junctions from this   }
{ strand, and cross over to the ignored strand.               }
{ NB at this point there may be no more junctions left so be careful}

    continue := true;{ we must determine the state at our current position}
        { on the ignored strand before we cross over        }

    while (IgnoredJunctions <= IJlast) and continue do
      begin
        continue := (Rlessthanequals(jpos(junctionindex^^[IgnoredJunctions]),
cRec.positions[i]));{?}
        if continue then
          begin
            ignored_state := nextstate(junctionindex^^[IgnoredJunctions]);{ junction=> change of
state }
            IgnoredJunctions := IgnoredJunctions + 1
          end;
      end;
{ There are now either no more junctions on this strand, or        }
{ IgnoredJunctions points to the junction AFTER the chiasma, and          }
{ Ignored_state indicates the state COMMING UP TO that junction. }
{ Either way we now wish to copy from this strand.              }
{ NB at this point there may be no more junctions left so be careful}

    if (read_state <> ignored_state) then{ Chiasma produces..}
      begin
        crash := (newpool.size = 0);
        if crash then
          goto 999;           {..a CRASH ..or..}
        junctionindex^^[newpool.start] := tuple(cRec.positions[i], ignored_state); {...a junction}
          {?}
        NEW_JUNCTION(newstrand);
          {?}
        temp_state := ignored_state; {AFTER the chiasmata}
        if ReadJunctions <= RJlast then
    { if there were junctions on the read strand }
          if cRec.positions[i] = jpos(junctionindex^^[ReadJunctions]) then
      { if there was a junction AT the chiasmata }
            read_state := nextstate(junctionindex^^[ReadJunctions]);
        { then we must change the read state to the state AFTER the junction }

        ignored_state := read_state;  {AFTER the chiasmata }
        read_state := temp_state;     { junction=>change of state }
      end;
 {we now cross over}
    temp := IgnoredJunctions;
    IgnoredJunctions := ReadJunctions;
    ReadJunctions := temp;
    temp := IJlast;
    IJlast := RJlast;
    RJlast := temp
  end;{chiasmata loop}

 { We now must transcribe any remaining junctions after the last chiasma}
  while (ReadJunctions <= RJlast) do
    begin
      crash := (newpool.size = 0);
      If crash then
```

```
          goto 999;
        junctionindex^^[newpool.start] := junctionindex^^[ReadJunctions];
        NEW_JUNCTION(newstrand);
        ReadJunctions := ReadJunctions + 1
      end;
999:
  end;{RECOMBINE}
{______________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure emptybuckets (var bucket, conts: bucketrack);
{-----------------------------------------------------------------}
    var
      i: integer;
  begin
    for i := 1 to nbuckets do
      begin
        bucket[i].count := 0;{?}
        conts[i].count := 0;{?}
      end;
  end;
{________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure zeroConts (var contributions: contrack);
{-----------------------------------------------------------------}
    var
      i: integer;
  begin
    for i := 1 to contributers do
      contributions[i] := 0;{?}
  end;
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure setbuckets (var bucket, conts: bucketrack;
                  n: integer);
{-----------------------------------------------------------------}
    var
      i, j: integer;
  begin
    bucket[1].top := 1.0;
    for i := 2 to nbuckets do
      bucket[i].top := bucket[i - 1].top * 0.8;
    conts[1].top := n;
    for i := 2 to nbuckets do
      conts[i].top := conts[i - 1].top * 0.8;
  end;
{________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function TRUE_STATE (haplo: strand;
                  qstate: state;
                  nchromosomes: integer): real;{?}
{-----------------------------------------------------------------}
{ The number retured is  the fraction of  }
{ the strand that is in the state qstate }
{??? with the advent of multiple states this procedure inherently becomes very}
{inneficient!!!}
```

```
{_____________________________________________________________}
  var
    strand_state: state;{?}
    j: Longint;
    total, marker: real;
    subject: strand;
    over: boolean;

begin
  subject := haplo;
  j := subject.start;

  over := false;
  while (j <= subject.last) and not over do
    if (jpos(junctionindex^^[j]) < 1 / nchromosomes) then
      j := j + 1
    else
      over := true;
  if over then
    subject.last := j - 1;

  marker := 0;
  total := 0;
  strand_state := subject.start_state;
  j := subject.start;

  while (j <= subject.last) do{ if there any junctions}
    begin
      if strand_state = qstate then
        total := total + (jpos(junctionindex^^[j]) - marker)
      else
        marker := jpos(junctionindex^^[j]);
      strand_state := nextstate(junctionindex^^[j]);{?}
      j := j + 1
    end;
  if strand_state = qstate then {?}
    total := total + ((1 / nchromosomes) - marker);
  if total > 0.001 + 1 / nchromosomes then
    with subject do
      begin
        writeln(start, ' ', last, ' ', SID, ' ', MSID, ' ', FSID);
        writeln(start_state);
        if start <= last then
          for j := start to last do
            writeln(jpos(junctionindex^^[j]), ' ', nextstate(junctionindex^^[j]));
        total := total + 1000;
      end;
  TRUE_STATE := total * nchromosomes;
end;{TRUE_STATE}
{_____________________________________________________________}
{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
function BINARY_ONECOUNT (subject: strand;
            nbases: LongInt): integer;
{---------------------------------------------------------------------}
{ The number returned is  the number of 1s in the binary pool corresponding to }
{ the sections of original genome which make up the strand }
{_____________________________________________________________}
  var
    strand_state: state;
    total, j: Longint;
    marker: real;
```

```
   function bconv (A: real): integer;
{ converts reals on a strand to integers to access the binary array }
{ for the moment, strands starts in the binary array are consecutive by their state no}
   begin
     bconv := strand_state + round(nbases * A);
   end;

 begin
   marker := 0;
   total := 0;
   strand_state := subject.start_state;
   j := subject.start;

   while (j <= subject.last) do{ if there any junctions}
     begin
       total := total + TRUE_TOTAL(bconv(marker), bconv(jpos(junctionindex^^[j])));
       marker := jpos(junctionindex^^[j]);
       strand_state := nextstate(junctionindex^^[j]);{?}
       j := j + 1
     end;
   total := total + TRUE_TOTAL(bconv(marker), bconv(1));
   BINARY_ONECOUNT := total
 end;{BINARY_ONECOUNT }
{___________________________________________________________________________}


 {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure putinabucket (var bucket: buckettrack;
                 blocksize: real);
 {--------------------------------------------------------------------------}
   var
     i: integer;
 begin
   i := 1;
   while (blocksize <= bucket[i].top) and (i < nbuckets) do{?}
     i := i + 1;
   bucket[i].count := bucket[i].count + 1{?}
 end;
 {___________________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function bucketCOUNT_STATE (var contributions: contrack;
                 var bucket: buckettrack;
                 subject: strand;
                 justzeroes: boolean): integer;
{--------------------------------------------------------------------------}
{ The number retured is  the number of junctions on the strand}
{as a side effect the distribution of blocks in the population }
{and the total contribution of each state to the population is worked out}
{if justzeroes then only blocks of state one are used for distribution}
{___________________________________________________________________________}
   var
     strand_state: state;{?}
     j: Longint;
     total, marker, templ: real;
     count: integer;

 {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
   procedure addtoContributions (thestate: state;
                 blocksize: real);
```

```
{-------------------------------------------------------------------}
  begin
    contributions[thestate] := contributions[thestate] + blocksize;
  end;
{___________________________________________________________________}

begin
  marker := 0;
  total := 0;
  strand_state := subject.start_state;
  j := subject.start;

  while (j <= subject.last) do
    begin
      templ := (jpos(junctionindex^^[j]) - marker);{?}
      addtocontributions(strand_state, templ);{?}
      if justzeroes then
        begin
          if strand_state = 0 then
            putinabucket(bucket, templ);{?}
        end
      else
        putinabucket(bucket, templ);{?}
      marker := jpos(junctionindex^^[j]);{?}
      strand_state := nextstate(junctionindex^^[j]);{?}
      j := j + 1
    end;
  templ := 1 - marker;{?}
  addtocontributions(strand_state, templ);{?}
  if justzeroes then
    begin
      if strand_state = 0 then
        putinabucket(bucket, templ);{?}
    end
  else
    putinabucket(bucket, templ);{?}
  if subject.start <= subject.last then
    bucketCOUNT_STATE := subject.last - subject.start + 1
  else
    bucketCOUNT_STATE := 0;
end;{bucketTRUE_STATE}
{___________________________________________________________________}

{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
function DOMINANCE (haplotype1, haplotype2: strand;
                 nchromosomes: integer): real;
{-------------------------------------------------------------------}
{ compares the two strands and calculates the proportion which has at least one 1 allele}
{___________________________________________________________________}
  var
    DOMstate, sameplace, over: boolean;
    i, j, k, klast: Longint;
    total, marker, temppos, pos1, pos2: real;
    state1, state2, otherstate: state;
    subject1, subject2: strand;


begin
  if (haplotype1.last = haplotype1.start + 1) and (haplotype2.last = 1 + haplotype2.start) then
    begin
    end;
```

```
    subject1 := haplotype1;
    subject2 := haplotype2;
    i := subject1.start;
    j := subject2.start;
    over := false;
    while (i <= subject1.last) and not over do
      if (jpos(junctionindex^^[i]) < 1 / nchromosomes) then
        i := i + 1
      else
        over := true;
    if over then
      subject1.last := i - 1;
    over := false;
    while (j <= subject2.last) and not over do
      if (jpos(junctionindex^^[j]) < 1 / nchromosomes) then
        j := j + 1
      else
        over := true;
    if over then
      subject2.last := j - 1;

    marker := 0;
    total := 0;
    state1 := subject1.start_state;
    state2 := subject2.start_state;
    domstate := (state1 = 1) or (state2 = 1);
    i := subject1.start;
    j := subject2.start;
{marker and states are set up, and domstate is set}

 {The first case is when both strands have junctions}
    while (i <= subject1.last) and (j <= subject2.last) do
      begin
        pos1 := jpos(junctionindex^^[i]);
        pos2 := jpos(junctionindex^^[j]);
        sameplace := pos1 = pos2;
        if pos1 <= pos2 then
          begin
            temppos := pos1;
            state1 := nextstate(junctionindex^^[i]);
            i := i + 1;
          end
        else
          begin
            temppos := pos2;
            state2 := nextstate(junctionindex^^[j]);
            j := j + 1;
          end;
        if sameplace then
          begin
            state2 := nextstate(junctionindex^^[j]);
            j := j + 1;
          end;
        if domstate then
          total := total + (temppos - marker)
        else
          marker := temppos;
        domstate := (state1 = 1) or (state2 = 1);
      end;
{one or both have reached their last junction,and it has been processed}
{marker and states are set up, and hetstate is set}
```

```
{The second case is ONE strand has no (more) junctions, so we must concentrate on the other }
    k := -1;
    if i <= subject1.last then
      begin
        k := i;
        klast := subject1.last;
        otherstate := state2;
      end
    else if j <= subject2.last then
      begin
        k := j;
        klast := subject2.last;
        otherstate := state1;
      end;
{The k variables now work on the important strand, and other state is constant for the }
{other strand as it has no more junctions }
{STILL: marker and states are set up, and hetstate is set}

{ Still on the second case ...}
    if k <> -1 then
      while (k <= klast) do
        begin
          if domstate then
            total := total + (jpos(junctionindex^^[k]) - marker)
          else
            marker := jpos(junctionindex^^[k]);
          domstate := (nextstate(junctionindex^^[k]) = 1) or (otherstate = 1);
          k := k + 1
        end;

{The third and final case is that there are no (more) junctions on either strand}
    if domstate then
      total := total + ((1 / nchromosomes) - marker);

    DOMINANCE := TOTAL * nchromosomes;
{   if TOTAL <> 0 then }
{   writeln(TOTAL * nchromosomes : 3 : 2); }
  end;{DOMINANCE}
{________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function HETEROZYGOSITY (haplotype1, haplotype2: strand;
                  nchromosomes: integer;
                  first_chr: boolean): real;
{--------------------------------------------------------------------}
{ compares the two strands and calculates the proportion which is heterozygous}
{__________________________________________________________________}
    var
      hetstate, sameplace, over: boolean;
      i, j, k, klast: Longint;
      total, marker, temppos, pos1, pos2: real;
      state1, state2, otherstate: state;
      subject1, subject2: strand;


    procedure select_chr (var subject: strand;
                   first_chr: boolean);
      var
        over: boolean;
        i: LongInt;
```

```
        thestate: state;
    begin
      i := subject.start;
      over := false;
      thestate := subject.start_state;
      if first_chr then
        while (i <= subject.last) and not over do
          if (jpos(junctionindex^^[i]) < 1 / nchromosomes) then
            i := i + 1
          else
            over := true
      else { not first_chr }
        while (i <= subject.last) and not over do
          begin
            temppos := jpos(junctionindex^^[i]);
            if (RlessThanEquals(jpos(junctionindex^^[i]), 1 / nchromosomes)) then { care must be
taken for real equalities }
              begin
                thestate := nextstate(junctionindex^^[i]);
                i := i + 1
              end
            else
              over := true;
          end;

      if first_chr then
        begin
          if over then
            subject.last := i - 1
        end
      else { not first_chr }
        begin
          if not over then
            begin
              subject.last := 0; { need not bother with any of the junctions }
              subject.start_state := thestate;
            end
          else { OVER! }
            begin
              if subject.last - subject.start >= 1 then
                begin
                  subject.start_state := nextstate(junctionindex^^[i - 1]); { else it just stays the same
}
                end;
              subject.start := i;
            end;
        end;

    end;{select_chr}


  begin
    subject1 := haplotype1;
    subject2 := haplotype2;
    i := subject1.start;
    j := subject2.start;


    select_chr(subject1, first_chr);
    select_chr(subject2, first_chr);
```

```
    if first_chr then
      marker := 0
    else
      marker := 1 / nchromosomes;
    total := 0;
    state1 := subject1.start_state;
    state2 := subject2.start_state;
    hetstate := state1 <> state2;
    i := subject1.start;
    j := subject2.start;
{marker and states are set up, and hetstate is set}

 {The first case is when both strands have junctions}
    while (i <= subject1.last) and (j <= subject2.last) do
      begin
        pos1 := jpos(junctionindex^^[i]);
        pos2 := jpos(junctionindex^^[j]);
        sameplace := Requals(pos1, pos2);{ care must be taken for real equalities }
        if Rlessthanequals(pos1, pos2) then
          begin
            temppos := pos1;
            i := i + 1;
            if (i <= subject1.last) then
              state1 := nextstate(junctionindex^^[i]);
          end
        else
          begin
            temppos := pos2;
            j := j + 1;
            if (j <= subject2.last) then
              state2 := nextstate(junctionindex^^[j]);
          end;
        if sameplace then
          begin
            j := j + 1;
            if (j <= subject2.last) then
              state2 := nextstate(junctionindex^^[j]);
          end;
        if hetstate then
          begin
            total := total + (temppos - marker);
          end;
        hetstate := state1 <> state2;
        if hetstate then
          marker := temppos;
      end;
{one or both have reached their last junction,and it has been processed}
{marker and states are set up, and hetstate is set}

{The second case is ONE strand has no (more) junctions, so we must concentrate on the other }
    k := -1;
    if i <= subject1.last then
      begin
        k := i;
        klast := subject1.last;
        otherstate := state2;
      end
    else if j <= subject2.last then
      begin
        k := j;
```

```
        klast := subject2.last;
        otherstate := state1;
      end;
{The k variables now work on the important strand, and other state is constant for the }
{other strand as it has no more junctions }
{STILL: marker and states are set up, and hetstate is set}

{ Still on the second case ...}
    if k <> -1 then
      while (k <= klast) do
        begin
          if hetstate then
            total := total + (jpos(junctionindex^^[k]) - marker)
          else
            marker := jpos(junctionindex^^[k]);
          hetstate := nextstate(junctionindex^^[k]) <> otherstate;
          k := k + 1
        end;

{The third and final case is that there are no (more) junctions on either strand}
    if hetstate then
      begin
        if first_chr then
          total := total + ((1 / nchromosomes) - marker)
        else
          total := total + (1 - marker)
      end;
    if ((total * 3) - round(total * 3) > 0.01) or ((total * 3) > 2.1) then
      begin
      end;
    if first_chr then
      HETEROZYGOSITY := TOTAL * nchromosomes
    else
      HETEROZYGOSITY := TOTAL / (1 - 1 / nchromosomes);
  end;{HETEROZYGOSITY}
{____________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function bucketHETEROZYGOSITY (var bucket: buckettrack;
                    haplotype1, haplotype2: strand;
                    nchromosomes: integer): real;
{--------------------------------------------------------------------------}
{ compares the two strands and calculates the proportion which is heterozygous}
{ dumps block sizes at the same time somehow!!}
{______________________________________________________________}
    var
      hetstate, sameplace, onestate, twostate, done: boolean;
      i, j, k, klast: Longint;
      total, marker, temppos, onemarker, twomarker, pos1, pos2: real;
      state1, state2, otherstate: state;
      subject1, subject2: strand;


  begin
    subject1 := haplotype1;
    subject2 := haplotype2;
    i := subject1.start;
    j := subject2.start;
    while (i <= subject1.last) and not done do
      if (jpos(junctionindex^^[i]) < 1 / nchromosomes) then
```

```
        i := i + 1
      else
        done := true;
    subject1.last := i - 1;
    done := false;
    while (j <= subject2.last) and not done do
      if (jpos(junctionindex^^[j]) < 1 / nchromosomes) then
        j := j + 1
      else
        done := true;
    subject2.last := j - 1;

    marker := 0;
    total := 0;
    state1 := subject1.start_state;
    state2 := subject2.start_state;
    hetstate := state1 <> state2;
    onemarker := 0;
    twomarker := 0;
    i := subject1.start;
    j := subject2.start;
{marker and states are set up, and hetstate is set}

 {The first case is when both strands have junctions}
    while (i <= subject1.last) and (j <= subject2.last) do
      begin
        sameplace := pos1 = pos2;
        if pos1 <= pos2 then
          begin
            temppos := pos1;
            putinabucket(bucket, pos1 - onemarker);
            onemarker := temppos;
            i := i + 1;
            if (i <= subject1.last) then
              state1 := nextstate(junctionindex^^[i]);
          end
        else
          begin
            temppos := pos2;
            putinabucket(bucket, pos2 - twomarker);
            twomarker := temppos;
            j := j + 1;
            if (j <= subject2.last) then
              state2 := nextstate(junctionindex^^[j]);
          end;
        if hetstate then
          total := total + (temppos - marker)
        else
          marker := temppos;
        if sameplace then
          begin
            putinabucket(bucket, pos2 - twomarker);
            twomarker := pos2;
            j := j + 1
          end;
        state1 := nextstate(junctionindex^^[i]);
        state2 := nextstate(junctionindex^^[j]);
        hetstate := state1 <> state2;
      end;
{one or both have reached their last junction,and it has been processed}
{marker and states are set up, and hetstate is set}
```

```
{The second case is ONE strand has no (more) junctions, so we must concentrate on the other }
    k := -1;
    if i <= subject1.last then
      begin {the other one is finished, but may have a last block }
        putinabucket(bucket, 1 - twomarker);
        k := i;
        klast := subject1.last;
        otherstate := state2;
      end
    else if j <= subject2.last then
      begin{the other one is finished, but may have a last block }
        otherstate := state1;
        putinabucket(bucket, 1 - onemarker);
        state1 := state2;
        onemarker := twomarker;
        k := j;
        klast := subject2.last;
      end;
{The k variables now work on the important strand, and other state is constant for the }
{other strand as it has no more junctions }
{STILL: marker and states are set up, and hetstate is set}

{ Still on the second case ...}
    if k <> -1 then
      begin
        while (k <= klast) do
          begin
            if hetstate then
              total := total + (jpos(junctionindex^^[k]) - marker)
            else
              marker := junctionindex^^[k];
            hetstate := nextstate(junctionindex^^[k]) <> otherstate;
            putinabucket(bucket, junctionindex^^[k] - onemarker);
            onemarker := junctionindex^^[k];
            k := k + 1;
            putinabucket(bucket, 1 - onemarker);
          end;
      end;
    if hetstate then
      total := total + ((1 / nchromosomes) - marker);
    bucketHETEROZYGOSITY := TOTAL * nchromosomes;
  end;{bucketHETEROZYGOSITY}
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function PURE (subject: strand;
                 qstate: state): boolean;{?}
{-----------------------------------------------------------}
{ Is the strand pure state x? }
{_________________________________________________________________}
  begin
    pure := (subject.start_state = qstate) and (subject.last = 0);{?}
  end;{PURE}
{_________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function STATE_AT (gposition: real;
                 subject: strand): state;{?}
```

```
{----------------------------------------------------------------------}
{ The state of the strand at the point-gene specified is returned. If there   }
{ is a junction at this point then the state returned is minus 1.}
{?}
{?}
{?}
{______________________________________________________________________}
   var
     strand_state: state;
     reached: boolean;
     j: Longint;

 begin
   strand_state := subject.start_state;
   j := subject.start;
   reached := false;

   while (j <= subject.last) and not (reached) do
     begin
       reached := (Rlessthanequals(gposition, jpos(junctionindex^^[j])));{?}
       if reached then
         if jpos(junctionindex^^[j]) = gposition then{?}
           STATE_AT := -1{indeterminate}
         else
           STATE_AT := strand_state;
       strand_state := nextstate(junctionindex^^[j]);
       j := j + 1
     end;
   if not reached then
     STATE_AT := strand_state
 end;{STATE_AT}
{__________________________________________________________________}



{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function USED_RECENTLY: Longint;
{-----------------------------------------------------------------}
{ Returns the number of junctions used since the last inquiry, or t=0       }
{__________________________________________________________________}
 begin
   USED_RECENTLY := recent_junctions;
   recent_junctions := 0
 end;
{__________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure PRINT_STRAND (joe: strand;
                 x: integer);
{---------------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of strand      }
{  assuming there are less than 10 states!!!}
{?}
{______________________________________________________________________}
   var
     ReadJunctions: Longint;
     read_state: state;{?}
     marker: real;
     i, count, reps: integer;
```

```
  begin
    ReadJunctions := joe.start;
    read_state := joe.start_state;

    write('strand', x : 2, ' ');
    for i := 1 to text_loci do
      write('_');
    writeln;
    write('        |');

    marker := 0;
    count := 0;
    while (ReadJunctions <= joe.last) do
      begin
        reps := round((jpos(junctionindex^^[ReadJunctions]) - marker) * text_loci);{?}
        for i := 1 to reps do
          write(read_state : 1);{?}
        count := count + reps;
        read_state := nextstate(junctionindex^^[ReadJunctions]);   { junction=>change of state }
           {?}
        marker := jpos(junctionindex^^[ReadJunctions]);{?}
        ReadJunctions := ReadJunctions + 1
      end;

    for i := 1 to (text_loci - count) do
      write(read_state : 1);{?}

    writeln('|');

    for i := 1 to text_loci do
      write('"');
    writeln
  end;{PRINT_STRAND}
{______________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SAVE_STRAND (joe: strand);
{-------------------------------------------------------------}
{______________________________________________________________________}
  begin
    with joe do
      begin
        start := start - newpool.base;
        if last = 0 then
          last := -33
        else
          last := last - newpool.base;
      end;
    write(secret, joe);
  end;{SAVE_STRAND}
{______________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure LOAD_STRAND (var joe: strand);
{-------------------------------------------------------------}
{______________________________________________________________________}
  begin
    read(secret, joe);
    with joe do
```

```
    begin
      start := start + newpool.base;
      if last = -33 then
        last := 0
      else
        last := last + newpool.base;
    end;
  end;{LOAD_STRAND}
{_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure SAVE_NEWPOOL (destination: string);
{------------------------------------------------------------------}
    var
      i: Longint;
{_____________________________________________________________________}
  begin
    rewrite(bobbys, destination);
    write(bobbys, newpool.start - newpool.base);
    for i := newpool.base to newpool.start - 1 do
      begin
        write(bobbys, junctionindex^^[i]);
      end;
    close(bobbys);
  end;{SAVE_NEWPOOL}
{_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure LOAD_NEWPOOL;
{------------------------------------------------------------------}
    var
      i, tint: Longint;
      temp: real;
{_____________________________________________________________________}
  begin
    reset(bobbys, 'h.dls');
    read(bobbys, temp);
    tint := round(temp);
    newpool.start := newpool.base + tint;
    for i := newpool.base to newpool.start - 1 do
      begin
        read(bobbys, junctionindex^^[i]);
      end;
    close(bobbys);
  end;{LOAD_NEWPOOL}
{_____________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure REWRITE_SECRET (destination: string);
{------------------------------------------------------------------}
{_____________________________________________________________________}
  begin
    rewrite(secret, destination);
  end;{REWRITE_SECRET}
{_____________________________________________________________________}

{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RESET_SECRET;
```

```
{---------------------------------------------------------------------}
{_________________________________________________________________}
  begin
    reset(secret, 'h.oz');
  end;{RESET_SECRET}
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure CLOSE_SECRET;
{---------------------------------------------------------------------}
{_________________________________________________________________}
  begin
    CLOSE(secret);
  end;{CLOSE_SECRET}
{_________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure PRINT_CHIASMATA (cRec: chiasmataRec);
{---------------------------------------------------------------------}
{ Uses text characters and standard I/O to show representation of chiasmata   }
{_________________________________________________________________}
    var
      i, j: integer;
      marker: real;

  begin
    marker := 0;
    write('Chiasma: ');
    for i := 1 to (cRec.total) do
      begin
        for j := 1 to (round((cRec.positions[i] - marker) * text_loci) - 1) do
          write(' ');
        write('|');
        marker := cRec.positions[i]
      end;
    writeln
  end;{PRINT_CHIASMATA}
{_________________________________________________________________}

end.
```

# A2.1 Diploid unit

```
{SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
S}
{_______________________________________________________}
{| ST PROSPERO PASCAL => MAC THINK PASCAL CONVERSION:  |( Stuart Baird 24/10/91
)}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
S}

unit Diploid;
```

## interface

```
  uses
    Junctions;


  const
    infinitedememax = 15000;
    infinity = -1;
    maxgenes = 100;
    maxdemes = 40;
    maxinds = 10000;
    dummy0 = 0;
    dummy1 = 100;
    checkseln = false;
    checkmigrn = false;
    logitmax = 5;{bounds on the logit function}

  type
    bittype = BOOLEAN;
    haplotype = STRAND;
    diplotype = record
        mt, pt: haplotype;{ stands for matrilineal and patrilineal NOT to be confused as Mother/Male,
Father/Female}
        ml, pl: boolean;{ matrilineal and patrilinealy inherited }
      end;
    genetype = -1..maxgenes;
    demetype = 1..maxdemes;
    indtype = LongInt;

    demearray = array[1..infinitedememax] of diplotype;
    demearrayptr = ^demearray;

    sexpoptype = array[1..maxdemes] of demearrayptr;
    sexpopptrtype = ^sexpoptype;
    poptype = record
        m, f: sexpopptrtype;
      end;
    gender = (male, female);


    samplearray = array[0..dummy0] of real;
    samplearrayptr = ^samplearray;

    dtdemetype = array[0..dummy0] of samplearrayptr;
    dtdemeptrtype = ^dtdemetype;
```

```
    dttype = array[0..dummy0] of real;
    dtptrtype = ^dttype;

    smallrealpvectype = array[0..dummy0] of real; { for haploids }
    smallrealpvecptrtype = ^smallrealpvectype;

    outarraytype = array[1..dummy1, 1..dummy1] of real;
    outarrayptrtype = ^outarraytype;

    posarray = array[1..20] of real;

    sexstuff = record
        sfunction: integer;
        s, B: real;
        ninds, nmig, nmigbarr: longint;
        mig: real;
        cumwptr: smallrealpvecptrtype;
      end;

  var
    bucket, conts: bucketrack;
    contributions: contrack;
    demefixed: array[1..maxdemes] of boolean;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure putdip (var p: sexpopptrtype;
                h2: diplotype;
                d: demetype;
                i: indtype);
{-------------------------------------------------------------------}
  procedure savedip (var p: sexpopptrtype;
                d: demetype;
                i: indtype);
{-------------------------------------------------------------------}
  procedure loaddip (var p: sexpopptrtype;
                d: demetype;
                i: indtype);
{-------------------------------------------------------------------}
  procedure swappop (var p, np: poptype);
{-------------------------------------------------------------------}
  function search (var t: smallrealpvecptrtype;
                x: real;
                nn: Longint): Longint;
{-------------------------------------------------------------------}
  procedure swapind (var a, b: indtype);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure transfer (var sexpop: sexpopptrtype;
                var newsexpop: sexpopptrtype;
                mfromleft, mfromright, mtoleft, mtoright: longint;
                demeno: demetype;
                ninds: Longint);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure migrate (var pop, newpop: sexpopptrtype;
                nmig, nmigbarr, ndemes: integer;
                ninds: LongInt;
                nmidleft, nmidright: integer);
{-------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure adjustdemes (var pop, newpop: sexpopptrtype;
```

```
                var ndemes, ninds: LongInt;
                var travel: integer;
                stretch: boolean;
                allZero: diplotype);
{-----------------------------------------------------------------}
{ added 8/2/93: Expanding Universe, for wave of advance }
{ We must always have an empty deme at the RHS to recieve the leading edge }
{ NB demefixed checks demes of both sexes, so that they can't be seperated}
{____________________________________________________________________________}
  procedure immigrate (var sexpop: sexpopptrtype;
                nmigbarr, ndemes: integer;
                allONE: diplotype);
{-----------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure double_immigrate (var sexpop: sexpopptrtype;
                nmig, nmigbarr, ndemes: integer;
                allONE, allZero: diplotype);
{-----------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_immigrate (var sexpop: sexpopptrtype;
                nmig, nmigbarr, ndemes: integer;
                allONE, allZero: diplotype;
                var ninds: longint);
{-----------------------------------------------------------------}
```

## implementation

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure putdip (var p: sexpopptrtype;
                  h2: diplotype;
                  d: demetype;
                  i: indtype);
{----------------------------------------------------------------}
 begin
   p^[d]^[i] := h2
 end;
{______________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure savedip (var p: sexpopptrtype;
                  d: demetype;
                  i: indtype);
{----------------------------------------------------------------}
 begin
   save_strand(p^[d]^[i].mt);
   save_strand(p^[d]^[i].pt);
 end;
{______________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure loaddip (var p: sexpopptrtype;
                  d: demetype;
                  i: indtype);
{----------------------------------------------------------------}
 begin
   load_strand(p^[d]^[i].mt);
   load_strand(p^[d]^[i].pt);
 end;
{______________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure swappop (var p: poptype;
                  var np: poptype);
{--------------------------------------------------------------}
   var
     tptr: poptype;
 begin
   tptr := p;
   p := np;
   np := tptr;
 end;
{____________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 function search (var t: smallrealpvecptrtype;
                  x: real;
                  nn: Longint): Longint;
{--------------------------------------------------------------}
{Looks for the position of x in a table of reals;t^[0]=0,t^[j]-t^[j-1]=w[j].}
```

```
{0<x<t^[imax]; returns j if t^[j-1]<x<=t^[j]}
{_______________________________________________________________}
   var
     i, imax, imin: Longint;
 begin
   if nn <= 1 then
     search := 1
   else
     begin
       imax := nn;
       imin := 0;{set the initial interval; x must lie between}
{                                   t^[imax] and t^[imin]}
       repeat
         i := (imax + imin) div 2;
         if x > t^[i] then
           imin := i
         else
           imax := i
       until imax = imin + 1;
       if x = t^[imin] then
         search := imin
       else
         search := imax
     end
 end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure swapind (var a, b: indtype);
{-----------------------------------------------------------------}
   var
     z: indtype;
 begin
   z := a;
   a := b;
   b := z
 end;
{_______________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure transfer (var sexpop: sexpopptrtype;
                  var newsexpop: sexpopptrtype;
                  mfromleft, mfromright, mtoleft, mtoright: longint;
                  demeno: demetype;
                  ninds: Longint);
{-----------------------------------------------------------------}
   var
     leftdeme, thisdeme, rightdeme: demearrayptr;
     offset: integer;
     i: indtype;
 begin
   if mfromleft > 0 then
     leftdeme := sexpop^[demeno - 1];
   thisdeme := sexpop^[demeno];
   if mfromright > 0 then
     rightdeme := sexpop^[demeno + 1];

   if mfromleft + mfromright <> mtoleft + mtoright then
     writeln('ERROR in transfer: asymmetric migration');
```

```
  if mfromleft > 0 then
    for i := 1 to mfromleft do
      putdip(newsexpop, leftdeme^[i], demeno, i);
  if mfromright > 0 then
    for i := ninds downto ninds - mfromright + 1 do
      putdip(newsexpop, rightdeme^[i], demeno, i);
  offset := mtoright - mfromleft;
  for i := mfromleft + 1 to ninds - mfromright do
    putdip(newsexpop, thisdeme^[i], demeno, i);
 end;
{______________________________________________________________}


{""""""""[sublevel of RTN 3 p8 ]""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure migrate (var pop, newpop: sexpopptrtype;
                  nmig, nmigbarr, ndemes: integer;
                  ninds: LongInt;
                  nmidleft, nmidright: integer);
{--------------------------------------------------------------}
{______________________________________________________________}
  var
    d: demetype;
    i, k, kb: indtype;
 begin
  k := round(nmig / 2);
  kb := round(nmigbarr / 2);
  if ndemes = 2 then
    begin
      transfer(pop, newpop, 0, kb, 0, kb, 1, ninds);
      transfer(pop, newpop, kb, 0, kb, 0, 2, ninds);
    end;
  if ndemes = 3 then
    begin
      transfer(pop, newpop, 0, kb, 0, kb, 1, ninds);
      transfer(pop, newpop, kb, k, kb, k, 2, ninds);
      transfer(pop, newpop, k, 0, k, 0, 3, ninds);
    end;
  if ndemes > 3 then
    begin
      transfer(pop, newpop, 0, k, 0, k, 1, ninds);
      if nmidleft - 1 > 1 then
        for d := 2 to nmidleft - 1 do
          transfer(pop, newpop, k, k, k, k, d, ninds);
      if nmidright + 1 < ndemes then
        for d := nmidright + 1 to ndemes - 1 do
          transfer(pop, newpop, k, k, k, k, d, ninds);
      transfer(pop, newpop, k, 0, k, 0, ndemes, ninds);
      transfer(pop, newpop, kb, k, kb, k, nmidright, ninds);
      transfer(pop, newpop, k, kb, k, kb, nmidleft, ninds)
    end;
 end;
{______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure adjustdemes (var pop, newpop: sexpopptrtype;
                  var ndemes, ninds: LongInt;
                  var travel: integer;
                  stretch: boolean;
```

```
                allZero: diplotype);
{---------------------------------------------------------------------}
{ added 8/2/93: Expanding Universe, for wave of advance }
{ We must always have an empty deme at the RHS to recieve the leading edge }
{ NB demefixed checks demes of both sexes, so that they can't be seperated}
{_______________________________________________________________________}
   var
     gain: integer; { The change in the number of demes }
     expand: boolean; { create a virgin deme }
     start, fin: integer; { The limits of the demes to be copied from the old pop}
     d, i: integer; {Deme and individual counters}
  begin
    gain := 0;
    start := 1;
    fin := ndemes;
    expand := false;
{First contract the start }
    if not stretch then { stretch implies we don't want the start to move }
      while demefixed[start] and (ndemes + gain > 2) do
        begin { Ignore it - it is now infinite }
          gain := gain - 1;
          start := start + 1;
        end;
{We may need to contract the end}
    while (demefixed[fin]) and (demefixed[fin - 1]) and (fin > 2) do
      begin
        gain := gain - 1;
        fin := fin - 1;
      end;
{Or we may wish to expand it with a virgin deme}
    if not demefixed[fin] then
      begin
        gain := gain + 1;
        expand := true
      end;
{Now do the actual copying}
    for d := start to fin do
      for i := 1 to ninds do
        putdip(newpop, pop^[d]^[i], d - start + 1, i);
{and expansion}
    ndemes := ndemes + gain;
    if expand then
      if ndemes > maxdemes then
        begin
          writeln('Whoa - deme overflow man! abort');
        end
      else
        for i := 1 to ninds do
          putdip(newpop, allZERO, ndemes, i);
    writeln('Travel= ', start - 1);
    travel := Travel + start - 1;
{Finished}
  end;
{_______________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure immigrate (var sexpop: sexpopptrtype;
                nmigbarr, ndemes: integer;
                allONE: diplotype);
```

```
{-----------------------------------------------------------------}
{_______________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    for d := 1 to ndemes do
      for i := 1 to nmigbarr do
        putdip(sexpop, allONE, d, i); { Pure invaders! }
{ NB no need to swap pops!}
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure double_immigrate (var sexpop: sexpopptrtype;
                  nmig, nmigbarr, ndemes: integer;
                  allONE, allZero: diplotype);
{-----------------------------------------------------------------}
{ added 9/7/92: prevents fixation in a finite population }
{_______________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    for d := 1 to ndemes do
      begin
        for i := 1 to nmigbarr do
          putdip(sexpop, allONE, d, i); { Pure invaders! }
        for i := nmigbarr + 1 to nmig + nmigbarr + 1 do
          putdip(sexpop, allZERO, d, i); { Pure natives! }
      end;{ NB no need to swap pops!}
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_immigrate (var sexpop: sexpopptrtype;
                  nmig, nmigbarr, ndemes: integer;
                  allONE, allZero: diplotype;
                  var ninds: longint);
{-----------------------------------------------------------------}
{_______________________________________________________________}
    var
      d: demetype;
      i, k, kb: indtype;
  begin
    for d := 1 to ndemes do
      for i := ninds + 1 to nmigbarr + ninds + 1 do
        putdip(sexpop, allONE, d, i); { Pure invaders! }
    ninds := ninds + nmigbarr;
{ NB no need to swap pops!}
  end;
{_______________________________________________________________}

end.
```

# A2.2 Stats unit

```
unit stats;
{24/11/93 changed to cope with sexual population }
interface
  uses
    General, Junctions, Diploid;
{graph;}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure INITIALISE_POSITIONS (nloci, nmarkergenes, nchromosomes: integer);
{----------------------------------------------------------------------}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure GET_CHROMOSOME_CHOICES (nchromosomes: integer;
                var choiceRec: chiasmataRec);
{----------------------------------------------------------------------}
{ eg 64 chromosomes so 64 (equally spaced) choices }
{________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure GET_CHIASMATA (nloci, nmarkergenes, nchromosomes: integer;
                mu: real;
                var cRec: chiasmataRec);
{----------------------------------------------------------------------}
{____________________________________________________________________}

{
  }
  function pointfitness (s: real;
                subject1, subject2: strand;
                qstate: state;
                B: real;
                sfunction, nmarkergenes: integer): real;
{----------------------------------------------------------------------}
{?}
{________________________________________________________________________}
  function fitness (s, x, B: real;
                sfunction: integer): real;
{- - - - - - - - - - - -  - - - - - - - - - - - - - - - - - - - - - - - - }
{______________________________________________________________________}
{
  }
  function getgenefreq (ninds: indtype;
                var d: demearrayptr;
                g: genetype;
                qstate: state): real;
{----------------------------------------------------------------------}
{ frequency of a gene in a particular deme                    }
{________________________________________________________________________}

{
  }
  function getgenediseq (ninds: indtype;
                var d: demearrayptr;
                g1, g2: genetype;
                var p1, p2, H: real;
                qstate: state): real;
{----------------------------------------------------------------------}
{ frequency of the genes in a particular deme, and returns the diseq between them }
{________________________________________________________________________}
    {^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^}
```

```
procedure find_extranuclear (ninds: indtype;
                pp: sexpopptrtype;
                d: demetype;
                var mt, pt: real);
{-----------------------------------------------------------------}
{ˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇ}
procedure find_wbar (s, B: real;
                sfunction: integer;
                ninds: indtype;
                nmarkergenes: genetype;
                nchromosomes: integer;
                pp: sexpopptrtype;
                d: demetype;
                var wb, varw, hetbar, varhet, hindbar, varhind: real;
                first_chr: boolean);
{-----------------------------------------------------------------}
{ see suzuki page 813                                             }
{_________________________________________________________________}
{ˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇ}
procedure bucketfind_wbar (sexvars: sexstuff;
                ninds: indtype;
                nmarkergenes: genetype;
                nchromosomes: integer;
                pp: sexpopptrtype;
                d: demetype;
                var wb, varw, hb, varh, ESone, ESzero: real;
                var onetot, zerotot: integer);
{-----------------------------------------------------------------}
{ see suzuki page 813                                             }
{_________________________________________________________________}
{
 }
procedure genefreqstats (ninds: indtype;
                nmarkergenes: genetype;
                var pp: sexpopptrtype;
                d: demetype;
                var genic_variance: real;
                var pbar: real);
{-----------------------------------------------------------------}
{_________________________________________________________________}
{
 }
function total_variance (ninds: indtype;
                nmarkergenes: genetype;
                var pp: sexpopptrtype;
                d: demetype): real;
{-----------------------------------------------------------------}
{_________________________________________________________________}
```

## implementation

```
 var
   positions: array[1..maxgenes] of real;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure INITIALISE_POSITIONS (nloci, nmarkergenes, nchromosomes: integer);
{-----------------------------------------------------------------}
   var
     i: integer;

 begin
   if nchromosomes = 1 then
     for i := 1 to nmarkergenes do
       positions[i] := (nmarkergenes - i + 0.5) / nmarkergenes
   else
     for i := 1 to nmarkergenes do
       positions[i] := (nmarkergenes - i + 0.5) / nchromosomes
 end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure GET_CHROMOSOME_CHOICES (nchromosomes: integer;
                 var choiceRec: chiasmataRec);
{-----------------------------------------------------------------}
{ eg 64 chromosomes so 64 (equally spaced) choices }
{_______________________________________________________________}
   var
     i: integer;
 begin
   choiceRec.total := 0;
   for i := 1 to nchromosomes do
     if tossAcoin then
       begin
         choiceRec.total := choiceRec.total + 1;
         choiceRec.positions[choiceRec.total] := (i - 1) / nchromosomes;
       end
 end;{GET_CHROMOSOME_CHOICES}
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure GET_CHIASMATA (nloci, nmarkergenes, nchromosomes: integer;
                 mu: real;
                 var cRec: chiasmataRec);
{------------[ sublevel of RTN 5, p10 ]------------------------------------}
{_______________________________________________________________}
   var
     i, x, pair, nmpairs, regionpairs, nchiasmata2, chtot: integer;
     region: real;
     c: char;
 begin
   if mu = 0 then { No recombination }
     cRec.total := 0
   else
     begin
       nmpairs := nmarkergenes - 1;
```

```
        chtot := 0;
        nchiasmata2 := Poisson(mu);    { Number of chiasmata on TWO chromatids }

        if nloci = infinity then
          begin
            for i := 1 to nchiasmata2 do
              cRec.positions[chtot + i] := pickAreal / nchromosomes;
{ only the first chromosome has infinite loci }
            chtot := chtot + nchiasmata2;
            BUBBLE_SORT(cRec.positions, chtot);
            cRec.total := chtot;
          end

        else { finite number of loci }
          begin
            cRec.total := 0;
            for i := 1 to nloci - 1 do
              if pickAreal <= mu then
                begin
                  cRec.total := cRec.total + 1;
                  cRec.positions[cRec.total] := (i / nloci) / nchromosomes;
{ only the first chromosome has n loci }
                end
          end;
      end;
  end;{GET_CHIASMATA}
{_____________________________________________________________________}


{
  }
  function fitness (s, x, B: real;
                  sfunction: integer): real;
{--------[sublevel of RTN 4, p9 ]----------------------------------------}
{______________________________________________________________}
  begin
    case sfunction of
      1:
        fitness := 1 - (s * x);              { additive: s<=f<=1 }
      2:
        fitness := 1 - exp(x * ln(s));           { 1-(s^x) : multiplicative: s<=f<=1 }
      3:
        fitness := 1 - s * (4 * x * (1 - x));  { epistatic: s<=f<=1 }
      4:
        fitness := 1 - s * exp(B * ln(4 * x * (1 - x))); { epistatic: Glaciated valley }
      5:
        fitness := 1 - s * (1 - x);            { additive: s<=f<=1, but invaders are fit }
                           { to model wave of advance }
      9:
        fitness := 1 + (s * x);              { additive: s<=1<=f }
      otherwise
        begin
          writeln('unspecified fitness function! ');
          repeat
            fitness := 0;
          until false
        end;
    end;
  end;
{______________________________________________________________}
```

```
{
  }
  function pointfitness (s: real;
                    subject1, subject2: strand;
                    qstate: state;
                    B: real;
                    sfunction, nmarkergenes: integer): real;
{-------------------------------------------------------------------}
{?}
{____________________________________________________________________}
    var
      alleles: 0..2;
      temp: real;
  begin
    case sfunction of
      9:
        begin
          alleles := 0;
          if STATE_AT(positions[nmarkergenes], subject1) = qstate then
            alleles := alleles + 1;
          if STATE_AT(positions[nmarkergenes], subject2) = qstate then
            alleles := alleles + 1;
          case alleles of
            0:
              pointfitness := 1;
            1:
              pointfitness := 1 + s / 2;
            2:
              pointfitness := 1 + s;
          end;
        end;
      10:
        begin
          temp := 1;
          if STATE_AT(positions[1], subject1) = qstate then
            temp := temp - 0.5;
          if STATE_AT(positions[1], subject2) = qstate then
            temp := temp - 0.5;
          if s * temp > 1 then
            pointfitness := 0
          else
            pointfitness := 1 - s * temp;
        end;
      11: {dominance}
        begin
          if (STATE_AT(positions[1], subject1) = qstate) or (STATE_AT(positions[1], subject2) =
qstate) then
            pointfitness := 1
          else
            pointfitness := 1 - s;
        end;
    end;
  end;
{____________________________________________________________________}


{
  }
  function getgenefreq (ninds: indtype;
```

```
                    var d: demearrayptr;
                    g: genetype;
                    qstate: state): real;
{---------------------------------------------------------------------}
{ frequency of a gene in a particular deme                            }
{?}
{_____________________________________________________________________}
   var
     sum: real;
     i: indtype;

  begin
   sum := 0;
   for i := 1 to ninds do
     begin
       if STATE_AT(positions[g], d^[i].mt) = qstate then
         sum := sum + 1;
       if STATE_AT(positions[g], d^[i].pt) = qstate then
         sum := sum + 1;
     end;
   getgenefreq := sum / (2 * ninds)
  end;{getgenefreq}
{_____________________________________________________________________}

{
 }
  function getgenediseq (ninds: indtype;
                    var d: demearrayptr;
                    g1, g2: genetype;
                    var p1, p2, H: real;
                    qstate: state): real;
{---------------------------------------------------------------------}
{ frequency of the genes in a particular deme, and returns the diseq between them }
{_____________________________________________________________________}
   var
     sum1, sum2, sumC, sumH: real;
     i: indtype;
     flagup: boolean;
     mstate1, mstate2, pstate1, pstate2: state;

  begin
   sum1 := 0;
   sum2 := 0;
   sumC := 0;
   sumH := 0;
   for i := 1 to ninds do
     begin
       flagup := false;
       mstate1 := STATE_AT(positions[g1], d^[i].mt);
       if mstate1 = qstate then
         begin
           sum1 := sum1 + 1;
           flagup := true;
         end;
       mstate2 := STATE_AT(positions[g2], d^[i].mt);
       if mstate2 = qstate then
         begin
           sum2 := sum2 + 1;
           if flagup then
             sumC := sumC + 1;
         end;
```

```
        flagup := false;
        pstate1 := STATE_AT(positions[g1], d^[i].pt);
        if pstate1 = qstate then
          begin
            sum1 := sum1 + 1;
            flagup := true;
          end;
        pstate2 := STATE_AT(positions[g2], d^[i].pt);
        if pstate2 = qstate then
          begin
            sum2 := sum2 + 1;
            if flagup then
              sumC := sumC + 1;
          end;

        if mstate1 <> pstate1 then
          sumH := sumH + 1;
        if mstate2 <> pstate2 then
          sumH := sumH + 1;
      end;
    p1 := sum1 / (2 * ninds);
    p2 := sum2 / (2 * ninds);
    H := sumH / (2 * ninds);
    getgenediseq := (sumC / (2 * ninds)) - (p1 * p2)
  end;{getgenediseq}
{_______________________________________________________________}

    {ˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇ}
  procedure find_effective_selection (joe: haplotype;
                  nmarkergenes: genetype;
                  qstate: state;
                  var zerocount, onecount: integer);
{--------------------------------------------------------------------}
{ Added  12:40pm 22/1/92  Stuart                                     }
{?}
{_______________________________________________________________}
    var
      g: integer;
  begin
    zerocount := 0;
    onecount := 0;
    for g := 1 to nmarkergenes do
      if STATE_AT(positions[g], joe) = qstate then
        onecount := onecount + 1
      else
        zerocount := zerocount + 1
  end;

    {ˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇˇ}
  procedure find_extranuclear (ninds: indtype;
                  pp: sexpopptrtype;
                  d: demetype;
                  var mt, pt: real);
{--------------------------------------------------------------------}
    var
      i: indtype;
  begin
    mt := 0;
    pt := 0;
    for i := 1 to ninds do
```

```
      begin
        if pp^[d]^[i].ml then
          mt := mt + 1;
        if pp^[d]^[i].pl then
          pt := pt + 1;
      end;
    mt := mt / ninds;
    pt := pt / ninds;
  end;
{______________________________________________________________}


  procedure find_wbar (s, B: real;
                  sfunction: integer;
                  ninds: indtype;
                  nmarkergenes: genetype;
                  nchromosomes: integer;
                  pp: sexpopptrtype;
                  d: demetype;
                  var wb, varw, hetbar, varhet, hindbar, varhind: real;
                  first_chr: boolean);
{------------------------------------------------------------}
{ see suzuki page 813                                         }
{______________________________________________________________}
    var
      i: indtype;
      ht, hi, smht, ssqht, smhi, ssqhi, w, smw, ssqw: real;
  begin
    smht := 0;
    ssqht := 0;
    smhi := 0;
    ssqhi := 0;
    smw := 0;
    ssqw := 0;
    for i := 1 to ninds do
      begin
{ the boolean flag on the end is first_chr }
        ht := HETEROZYGOSITY(pp^[d]^[i].mt, pp^[d]^[i].pt, nchromosomes, first_chr);
        hi := (true_state(pp^[d]^[i].mt, 1, nchromosomes) + true_state(pp^[d]^[i].pt, 1, nchromosomes))
/ 2;
        w := fitness(s, ht, B, sfunction);
        smht := smht + ht;
        ssqht := ssqht + sqr(ht);
        smhi := smhi + hi;
        ssqhi := ssqhi + sqr(hi);
        smw := smw + w;
        ssqw := ssqw + sqr(w);
      end;
    hetbar := smht / ninds;
    varhet := (ssqht / ninds) - sqr(hetbar);
    hindbar := smhi / (ninds);
    varHind := (ssqhi / ninds) - sqr(hindbar);
    wb := smw / ninds;
    varw := (ssqw / ninds) - sqr(wb);
  end;
{______________________________________________________________}

    {^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^}
  procedure bucketfind_wbar (sexvars: sexstuff;
                  ninds: indtype;
                  nmarkergenes: genetype;
```

```
                 nchromosomes: integer;
                 pp: sexpopptrtype;
                 d: demetype;
                 var wb, varw, hb, varh, ESone, ESzero: real;
                 var onetot, zerotot: integer);
{-----------------------------------------------------------------}
{ see suzuki page 813                                             }
{_________________________________________________________________}
   var
      i: indtype;
      h, smh, ssqh, w, smw, ssqw: real;
      zerocount, onecount: integer;
  begin
   with sexvars do
    begin
      smh := 0;
      ssqh := 0;
      smw := 0;
      ssqw := 0;
      onetot := 0;
      zerotot := 0;
      ESone := 0;
      ESzero := 0;
      for i := 1 to ninds do
       begin
         h := bucketHETEROZYGOSITY(bucket, pp^[d]^[i].MT, pp^[d]^[i].PT, nchromosomes);
         w := fitness(s, h, B, sfunction);
         smh := smh + h;
         ssqh := ssqh + sqr(h);
         smw := smw + w;
         ssqw := ssqw + sqr(w);
{?)  find_effective_selection(pp^[d]^[i], nmarkergenes, zerocount, onecount);}
{   ESone := ESone + (onecount * w);}
{   ESzero := ESzero + (zerocount * w);}
{   onetot := onetot + onecount;}
{   zerotot := zerotot + zerocount;}
       end;
      hb := smh / ninds;
      varh := (ssqh / ninds) - sqr(hb);
      wb := smw / ninds;
      varw := (ssqw / ninds) - sqr(wb);
      ESone := Esone / onetot;
      Eszero := ESzero / zerotot;
    end;
  end;
{_________________________________________________________________}


{`````````````````````````````````````````````}
 procedure genefreqstats (ninds: indtype;
                 nmarkergenes: genetype;
                 var pp: sexpopptrtype;
                 d: demetype;
                 var genic_variance: real;
                 var pbar: real);
{-----------------------------------------------------------------}
{_________________________________________________________________}
   var
      g: genetype;
      p, GVsum, Psum: real;
  begin
```

```
    GVsum := 0;
    Psum := 0;
    for g := 1 to nmarkergenes do
      begin
{?)  p := getgenefreq(ninds, pp^[d], g);}
{    GVsum := GVsum + p * (1 - p);   }
 { assuming haploidy }
{    Psum := Psum + p;}
      end;
    genic_variance := GVsum;
    pbar := Psum / nmarkergenes;
  end;
{______________________________________________________________}


{
  }
  function total_variance (ninds: indtype;
                    nmarkergenes: genetype;
                    var pp: sexpopptrtype;
                    d: demetype): real;
{---------------------------------------------------------------}
{______________________________________________________________}
    var
      k: indtype;
      g: genetype;
      p, sum, sum2, zind: real;
  begin
    sum := 0;
    sum2 := 0;
    for k := 1 to ninds do
      begin
        zind := 0;
        for g := 1 to nmarkergenes do
{?)  if STATE_AT(positions[g], pp^[d]^[k]) then}
{?)  zind := zind + 1;}
          sum := sum + zind;
        sum2 := sum2 + sqr(zind);
      end;
    total_variance := (sum2 / ninds) - sqr(sum / ninds);
                              { assuming haploidy }
  end;
{______________________________________________________________}

end.
```

# A2.4 Diploid Junctions program

```
program Diploid_junctions;
{ Created February/March 1991 by Stuart Baird,       }
{ using much code from the program:                   }
{ Multiloc                                            }
{ created June 1990: modified version of multloc9;    }
{ modified by Nick, September 1990                    }
{ modified by Nick, March    1991                     }
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{ 11/11/91 This now uses mymem for run-time arrays with THINK pascal }
{----------------------------------------------------------------------}
{ 22/1/92 This now outputs Effective selection pressures }
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
{ units of routines                                                      }
{----------------------------------------------------------------------}
{ 7/7/92 This will now simulate a single infinite deme                 }
{----------------------------------------------------------------------}
  uses
{graphics;}
    mymem, junctions, Diploid, general, stats, Binary;
{______________________________________________________________________}
  const
{   harddisc = 'Hard disc:Desktop Folder:Stuart:Stuarts Results:';}
{   harddisc = 'Quadra HD:Desktop Folder:Stuart:Stuarts Results';}
    harddisc = 'Macintosh HD:Desktop Folder:Stuart:Stuarts Results';
{   harddisc = 'RAM Disk';}

    standalone = true; {when set allows mouse to interrupt }
  label
    999, 333;

  type
    migration_styles = (no_migration, normal, waveofadvance, stretchwoa, Dwavestretchwoa,
from_infinite, double_from_infinite);
    basepop_styles = (infinitedeme, finite_deme_chain, seperate_mixed_demes, rare_allele);
    selection_styles = (art_selection, pointselection, normals, assortative, additive);
    output_styles = (ANCESTRY, MYHYBRIDS, boring, indices);
  var
    Masc, Femi: sexstuff;
    migration_style: migration_styles;
    basepop_style: basepop_styles;
    selection_style: selection_styles;
    output_style: output_styles;
    t, dt, sample, nsamples, travel: integer;
    ndemes, origndemes, nmidleft, nmidright: longint;
    nmarkergenes: genetype;
    nloci: genetype;
    nstates, nbases, nchromosomes: LongInt;

  var
    array_heap, pool_heap: Longint;
    run: string;
    out_file, out_Mgen, out_Fgen, out_Mdist, out_Fdist, out_mPbar, out_fPbar: Text;
    out_mWbar, out_mWvar, out_mHTvar, out_mHIvar, out_mHTbar, out_mHIbar, out_fWbar,
out_fWvar, out_fHTvar, out_fHIvar, out_fHTbar, out_fHIbar, context, bf: Text;
    in_file: file of integer;
    batch_fname, path: string;
    out_Mdistname, out_Fdistname, out_Mpbname, out_MHTvname, out_MHIvname,
out_Fpbname, out_fHTvname, out_fHIvname, out_distname: string;
```

```
    out_Mwbname, out_Mwvname, out_Mhtbname, out_MhIbname, out_Fwbname, out_Fwvname,
out_Fhtbname, out_FhIbname: string;
    control_string, TimeString, DateString: string;
    batch, selfrandom, done, crash, explode, pointselectionMale, sexual, stretch, Dwave: boolean;
    ans, ans2, ans3: string;
    pop, newpop: poptype;
    B, rec, mu, hwidth: real;
    replicates, repNo, twarm, tmax, runint: integer;
    seedNo, ninit, nfix, g: integer;
    temp1, temp2: LongInt;
    demeno: demetype;
    pbar, pvar, dbar: dtdemeptrtype;
    ip, iq, i1, i2, i3, i4, h: haplotype;
    d: demetype;
    i: indtype;

    demearraylength, poparraylength, pvecarraylength: Longint;

    windowrect: rect;
{ for opening folders 7/8/92 }
    error: OSerr;
    name: stringptr;
    creator, tipe: OStype;
    vol, unknownint: integer;
    unknownLongint, savednindsM, savednindsF: Longint;
    replicatesummary: string;
    completeruns, repstart, pausedthisrun, tstart: integer;
    precalculate: real;
    distribution, paused, breakoff: boolean;
    newpoint, oldpoint: point;
    allOne, allZero, allHet, RareOne: diplotype; {'pure' diplotypes}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infonlefttransfer (var pop: sexpopptrtype;
                    var newpop: sexpopptrtype;
                    mfromleft, mfromright, mtoleft, mtoright: longint;
                    demeno: demetype;
                    ninds: Longint);
{--------------------------------------------------------------------}
    var
      leftdeme, thisdeme, rightdeme: demearrayptr;
      offset: integer;
      i: indtype;
      h: haplotype;
  begin
    thisdeme := pop^[demeno];
    if mfromright > 0 then
      rightdeme := pop^[demeno + 1];
    if mfromleft > 0 then
      for i := 1 to mfromleft do
        putdip(newpop, allONE, demeno, i);
{PURE invaders! }
    if mfromright > 0 then
      for i := ninds downto ninds - mfromright + 1 do
        putdip(newpop, rightdeme^[i], demeno, i);
    offset := mtoright - mfromleft;
    for i := mfromleft + 1 to ninds - mfromright do
      putdip(newpop, thisdeme^[i], demeno, i);
  end;
{________________________________________________________________________}
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
```

```
procedure advance (var pop, newpop: sexpopptrtype;
                nmig, nmigbarr, ndemes: integer;
                ninds: LongInt;
                nmidleft, nmidright: integer);
{----------------------------------------------------------------------}
{ modified 25/5/94 for diffusion, brick wall now on left }
{ 0 migrants enter from infinite deme on the left. k migrants are transferred between all other demes }
{_____________________________________________________________________}
  var
    d: demetype;
    i, k, kb: indtype;
begin
  k := round(nmig / 2);
  kb := round(nmigbarr / 2);
  if ndemes = 1 then
    infonlefttransfer(pop, newpop, 0, 0, 0, 0, 1, ninds)
  else
    begin
      infonlefttransfer(pop, newpop, 0, k, 0, k, 1, ninds);
      for d := 2 to ndemes - 1 do
        transfer(pop, newpop, k, k, k, k, d, ninds);
      transfer(pop, newpop, k, 0, k, 0, ndemes, ninds);
    end;
end;
{_____________________________________________________________________}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
procedure select (pop: sexpopptrtype;
                sex: sexstuff;
                d: demetype;
                first: boolean);
{-------------[sublevel of RTN 4, p9 ]----------------------------------}
{ sets up a table of cumulative fitnesses for a deme> This is a crafty trick}
{ from multilocus - the subsequent search of this table will return an entry}
{ in proportion to the real "interval" it occupies.          }
{_____________________________________________________________________}
  var
    i: indtype;
    temp, stemp: real;
    tempbool, ok: boolean;
    tempsfunction: integer;
    selected_state: state;
    cutoff: LongInt;
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  function INSERT_SORT (A: smallrealpvecptrtype;
                n: integer): boolean;
{----------------------------------------------------------------------}
    var
      j: integer;
      temp: real;
      ok: boolean;

  begin
    ok := true;
    for j := 1 to n - 1 do
      if A^[j] = A^[n] then
        ok := false;
    if ok then
      for j := n downto 2 do
        if A^[j] < A^[j - 1] then
          begin
```

```
          temp := A^[j];
          A^[j] := A^[j - 1];
          A^[j - 1] := temp
        end;
      INSERT_SORT := ok
    end;{INSERT_SORT}
{_____________________________________________________________}
  begin
    selected_state := 1;
    with SEX do
      case selection_style of
        pointselection, assortative:
          begin
            cumwptr^[0] := 0;
            stemp := 0.2;
            tempsfunction := 5;
            for i := 1 to ninds do
              cumwptr^[i] := cumwptr^[i - 1] + fitness(stemp, HETEROZYGOSITY(pop^[d]^[i].mt,
pop^[d]^[i].pt, nchromosomes, true), B, tempsfunction);
            temp := cumwptr^[ninds];
            if migration_style = waveofadvance then
              tempbool := (abs(temp - ninds) < stemp) or (abs(temp - (ninds * (1 - stemp))) <
stemp);
            if first then
              demefixed[d] := tempbool
            else
              demefixed[d] := tempbool and demefixed[d];
            for i := 1 to ninds do { o individuals do less well, 1's are favoured }
              cumwptr^[i] := cumwptr^[i - 1] + pointfitness(s, pop^[d]^[i].mt, pop^[d]^[i].pt,
selected_state, B, sfunction, nmarkergenes);
          end;
        normals:
          begin
            cumwptr^[0] := 0;
            stemp := 0.2;
            for i := 1 to ninds do
              cumwptr^[i] := cumwptr^[i - 1] + fitness(s, HETEROZYGOSITY(pop^[d]^[i].mt,
pop^[d]^[i].pt, nchromosomes, true), B, sfunction);
            temp := cumwptr^[ninds];
            if migration_style = waveofadvance then
              tempbool := (abs(temp - ninds) < stemp) or (abs(temp - (ninds * (1 - stemp))) <
stemp);
            if first then
              demefixed[d] := tempbool
            else
              demefixed[d] := tempbool and demefixed[d];
          end;
        additive:
          begin
            cumwptr^[0] := 0;
            stemp := 0.2;
            for i := 1 to ninds do
              cumwptr^[i] := cumwptr^[i - 1] + fitness(s, (TRUE_STATE(pop^[d]^[i].mt, 1,
nchromosomes) + TRUE_STATE(pop^[d]^[i].pt, 1, nchromosomes)) / 2, B, sfunction);
            temp := cumwptr^[ninds];
            if migration_style = waveofadvance then
              tempbool := (abs(temp - ninds) < stemp) or (abs(temp - (ninds * (1 - stemp))) <
stemp);
            if first then
              demefixed[d] := tempbool
            else
```

```
              demefixed[d] := tempbool and demefixed[d];
          end;
        art_selection:
          begin
{ rank individuals by fitness }
            for i := 1 to ninds do
              begin
                cumwptr^[i] := BINARY_ONECOUNT(pop^[d]^[i].mt, nbases) +
BINARY_ONECOUNT(pop^[d]^[i].pt, nbases);
                ok := INSERT_SORT(cumwptr, i);
                if not ok then
                  begin
                  end;
              end;
            cumwptr^[0] := 0;
{chop off tail - a proportion B of the population are not used}
            cutoff := round(ninds * (1 - B));
            for i := 1 to cutoff do
              cumwptr^[i] := 0;
{ set up cumulant }
            for i := cutoff + 1 to ninds do
              cumwptr^[i] := cumwptr^[i - 1] + fitness(s, cumwptr^[i], B, sfunction);
          end;
      end;
  end;
{_____________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure randomparent (pop: sexpopptrtype;
                  sex: sexstuff;
                  d: demetype;
                  var mumordad: diplotype);
{--------------------------------------------------------------------}
{ returns individuals with equal probability }
{_____________________________________________________________________}
    var
      i1: LongInt;
      cumwptr: smallrealpvecptrtype;
  begin
    with sex do
      begin
        i1 := round(uniform * ninds + 0.5);
        if (i1 < 1) or (i1 > ninds) then
          writeln('oops !!');
        mumordad := pop^[d]^[i1];
      end;
  end;
{_____________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure parent (pop: sexpopptrtype;
                  sex: sexstuff;
                  d: demetype;
                  var mumordad: diplotype);
{ 29/6/92 infinite population variant }
{--------------------------------------------------------------------}
{ returns individuals with probability proportional to their fitness, using }
{ the table of cumulative fitness set in by select                          }
{_____________________________________________________________________}
```

```
  var
    i1: indtype;
    cumwptr: smallrealpvecptrtype;
 begin
  with sex do
   begin
    i1 := search(cumwptr, pickAreal * cumwptr^[ninds], ninds);
  {cumwptr^[i-1]<rand<=cumwptr^[i]}
     mumordad := pop^[d]^[i1];
   end;
 end;
{________________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure parents (mumpop, dadpop: sexpopptrtype;
                mumvars, dadvars: sexstuff;
                d: demetype;
                var mum, dad: diplotype);
{------------------------------------------------------------------}
{ returns individuals with probability proportional to their fitness, using }
{ the table of cumulative fitness set in by select                          }
{________________________________________________________________________}
  var
    i1, i2: indtype;

 begin
  with MUMVARS do
    i1 := search(Cumwptr, pickAreal * Cumwptr^[ninds], ninds);
  with DADVARS do
    i2 := search(Cumwptr, pickAreal * Cumwptr^[ninds], ninds);
  {cumwptr^[i-1]<rand<=cumwptr^[i]}
  mum := mumpop^[d]^[i1];
  dad := dadpop^[d]^[i2];
 end;
{________________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure reproduce (var pop: poptype;
                var newpop: poptype;
                rec: real);
{--------[sublevel of RTN 4, p9 ]-----------------------------------------}
  label
    999;

  var
    d, chNo, sink, UR: integer;
    jim: indtype;
    mum, dad, sonnyjim: diplotype;
    ovum, sperm, spermset1, ovumset1, spermset2, ovumset2: haplotype;
    chiasmata, chromosomes: chiasmataRec;
    c: char;
    totr, time: real;
    sex: gender;
    tempninds: longint;
    selected_state: state;
    success: boolean;
 begin
  selected_state := 1;
```

```
    time := t;
    for d := 1 to ndemes do
      begin
{set up the tables of cumulative fitnesses for this deme}
        demefixed[d] := true;
        select(pop.m, MASC, d, true);
        select(pop.f, FEMI, d, false);
        for sex := male to female do
          begin
            if sex = male then
              tempninds := MASC.ninds
            else
              tempninds := FEMI.ninds;
            for jim := 1 to tempninds do  { [ RTN 5, p10 ] }
              begin

{ AT THIS POINT ONLY, THE SEXES MaY INTERACT!!! }
                if sexual then
                  begin
                    if selection_style = assortative then
                      begin
                        success := true;
                        repeat
                          randomparent(pop.f, FEMI, d, mum);
{ get any mother }
                          if pointfitness(0.2, mum.mt, mum.pt, selected_state, B, 10, nmarkergenes)
< 0.9 then     { if mother is pure red }
                            randomparent(pop.m, MASC, d, dad)                    { father
can be anyone }
                          else                                          { otherwise }
                            begin
                              parent(pop.m, MASC, d, dad);                     { father
can't be pure red }
                              success := pointfitness(0.2, dad.mt, dad.pt, selected_state, B, 10,
nmarkergenes) > 0.8
                            end;
                        until success;
                      end
                    else
                      parents(pop.f, pop.m, FEMI, MASC, d, mum, dad);
                  end
                else {two seperate populations}
                  if sex = male then
                    parents(pop.m, pop.m, MASC, MASC, d, mum, dad)
                  else {sex=female}
                    parents(pop.f, pop.f, FEMI, FEMI, d, mum, dad);
{ THAT ALL THE INTERACTION FOLKS!! }

{ THE HAPLOID GAMETES GET EITHER MATERNAL OR PATERNAL CHR. TIDS }
                GET_CHROMOSOME_CHOICES(nchromosomes, chromosomes);
{ assumes equal chromosomes with infinite loci }
                RECOMBINE(crash, mum.mt, mum.pt, ovumset1, chromosomes);
                if crash then
                  goto 999;
                RECOMBINE(crash, mum.pt, mum.mt, ovumset2, chromosomes);
                if crash then
                  goto 999;
                GET_CHROMOSOME_CHOICES(nchromosomes, chromosomes);
{ assumes equal chromosomes with infinite loci }
                RECOMBINE(crash, dad.mt, dad.pt, spermset1, chromosomes);
                if crash then
```

```
                 goto 999;
               RECOMBINE(crash, dad.pt, dad.mt, spermset2, chromosomes);
               if crash then
                 goto 999;


{MEIOSIS to produce OVUM!!!}
{set up recomb. record, a sorted array of nchiasmata positions }
               GET_CHIASMATA(nloci, nmarkergenes, nchromosomes, mu, chiasmata);
{recombination produces haploid gamete}
               RECOMBINE(crash, ovumset1, ovumset2, ovum, chiasmata);
               if crash then
                 goto 999;
               if ovum.last <> 0 then
                 begin { stops in here }
                 end;
{MEIOSIS to produce SPERM!!!}
{set up recomb. record, a sorted array of nchiasmata positions }
               GET_CHIASMATA(nloci, nmarkergenes, nchromosomes, mu, chiasmata);
{recombination produces haploid gamete}
               RECOMBINE(crash, spermset1, spermset2, sperm, chiasmata);
               if crash then
                 goto 999;
               if sperm.last <> 0 then
                 begin { stops in here }
                 end;
{FUSION OF GAMETES}
               sonnyjim.mt := ovum;
               sonnyjim.pt := sperm;
               sonnyjim.ml := mum.ml;
               sonnyjim.pl := dad.pl;
{put the new individual into newpop}
               if sex = male then
                 putdip(newpop.m, sonnyjim, d, jim)
               else
                 putdip(newpop.f, sonnyjim, d, jim)
{record the state of the new deme if you are advancing }
             end;
         end;
     end;
999:
  end;
{______________________________________________________________}




{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure infinite_reproduce (var pop: poptype;
               var newpop: poptype;
               rec: real);
{ 29/6/92 infinite population variant }
{--------------[sublevel of RTN 4, p9 ]----------------------------------}
   label
     999;

   var
     d, chNo, sink: integer;
     jim: indtype;
     mumordad, sonnyjim: diplotype;
```

```
        gamete: haplotype;
        chiasmata: chiasmataRec;
        c: char;
        totr, time, wtot: real;
        offspring, x, y, z: integer;
        Mnew_ninds, Fnew_ninds, sexninds: longint;
        sex: gender;
        SEXVARS: sexstuff;
        sexpop, newsexpop, newothersexpop: sexpopptrtype;
  begin
    x := 0;
    y := 0;
    z := 0;
    wtot := 0;
    Mnew_ninds := 0;
    Fnew_ninds := 0;
    time := t;
    for d := 1 to ndemes do
      begin
{ don't set up the table of cumulative fitnesses for this deme}
{   select(pop, d);}
        for sex := male to female do
          begin
            if sex = male then
              begin
                sexvars := MASC;
                sexpop := pop.m;
                newsexpop := newpop.m;
                newothersexpop := newpop.f;
              end
            else
              begin
                sexvars := FEMI;
                sexpop := pop.f;
                newsexpop := newpop.f;
                newothersexpop := newpop.m;
              end;
            with SEXVARS do
              for jim := 1 to ninds do  { [ RTN 5, p10 ] }
                begin
{ SIGNIFICANTLY different from finite n case: }
{ infinite population => wbar==1 therefore fitness is nolonger relative to other individuals, }
{ but to 1. Cumulative table is therefore unnecessary }
{ Yet another major departure: Selection removes blocks length Y at a given rate in Nicks model, }
{ all individuals are now parents who produce Poisson(fitness) offspring in the next generation }
{ PROBLEM with conversion to 2 sex population:All fems and males will mate with natives }
{ how do we decide what sex their childer will be??: 50:50 determined just before birth **}
                  for offspring := 1 to Poisson(2 * fitness(s,
HETEROZYGOSITY(sexpop^[d]^[jim].mt, sexpop^[d]^[jim].pt, nchromosomes, true), B, sfunction))
do
                    begin
{MEIOSIS.. only matters in native }
                      GET_CHIASMATA(nloci, nmarkergenes, nchromosomes, mu, chiasmata);
{    if chiasmata.total = 0 then}
{     y := y + 1;}
{set up recomb. record, a sorted array of nchiasmata positions }
{ DON'T draw ONE parent, using the table of cumulative fitnesses set up by SELECT}
{    parent(pop, d, mumordad); }
{ALWAYS recombination between PURE NATIVE ( zero ) and recombinant/immigrant produces
haploid baby}
                      if tossAcoin then
```

```
                        RECOMBINE(crash, sexpop^[d]^[jim].mt, sexpop^[d]^[jim].pt, gamete,
chiasmata)
                    else
                        RECOMBINE(crash, sexpop^[d]^[jim].pt, sexpop^[d]^[jim].mt, gamete,
chiasmata);
                    if crash then
                        goto 999;
{FUSION OF GAMETES}
                    if sex = male then
                        begin
                            sonnyjim.pt := gamete;
                            sonnyjim.mt := AllZero.mt;{or pt}
                            sonnyjim.ml := AllZero.ml;
                            sonnyjim.pl := sexpop^[d]^[jim].pl;
                        end
                    else { sex=female}
                        begin
                            sonnyjim.mt := gamete;
                            sonnyjim.pt := AllZero.mt;{or pt}
                            sonnyjim.pl := AllZero.pl;
                            sonnyjim.ml := sexpop^[d]^[jim].ml;
                        end;

{put the new individual into newpop IF IT IS A RECOMBINANT}
{** nomatter what the sex of the parent, child has 50:50 chance of being female **}
                    if not (pure(sonnyjim.pt, 0) and pure(sonnyjim.mt, 0)) then
                        if tossAcoin then { new indiv is same sex }
                            begin
                                if sex = male then
                                    begin
                                        Mnew_ninds := Mnew_ninds + 1;
                                        sexninds := Mnew_ninds;
                                    end
                                else
                                    begin
                                        Fnew_ninds := Fnew_ninds + 1;
                                        sexninds := Fnew_ninds;
                                    end;
                                if (Mnew_ninds >= infinitedememax) or (Fnew_ninds >=
infinitedememax) then
                                    begin
                                        explode := true;
                                        goto 999;
                                    end;
                                putdip(newsexpop, sonnyjim, d, sexninds)
                            end
                        else { new indiv is the other sex }
                            begin
                                if sex = male then
                                    begin
                                        Fnew_ninds := Fnew_ninds + 1;
                                        sexninds := Fnew_ninds;
                                    end
                                else
                                    begin
                                        Mnew_ninds := Mnew_ninds + 1;
                                        sexninds := Mnew_ninds;
                                    end;
                                if (Fnew_ninds >= infinitedememax) or (Mnew_ninds >=
infinitedememax) then
                                    begin
```

```
                              explode := true;
                              goto 999;
                            end;
                          putdip(newothersexpop, sonnyjim, d, sexninds)
                        end;

                  end;
              end;
          end;
      end;
{record the new population size}
    with MASC do
      ninds := Mnew_ninds;
    with FEMI do
      ninds := Fnew_ninds;
999:
  end;
{_______________________________________________________________}




{''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''}
  procedure askstuff_batch;
{--------------------------------------------------------------------}
{_______________________________________________________________}
    var
      gg: genetype;
  begin
    readln(bf);
    readln(bf, replicatesummary);
    readln(bf);
    readln(bf, replicates, tmax, dt, twarm);
    readln(bf);
    readln(bf, ndemes, MASC.ninds, nstates, nloci, nbases, nmarkergenes, nchromosomes);
    FEMI.ninds := MASC.ninds;
    readln(bf);
    readln(bf, migration_style);
    Dwave := migration_style = Dwavestretchwoa;
    if (migration_style = stretchwoa) or (migration_style = Dwavestretchwoa) then
      begin
        stretch := true;
        migration_style := waveofadvance;
      end
    else
      stretch := false;
    readln(bf);
    readln(bf, basepop_style);
    readln(bf);
    readln(bf, selection_style);
    readln(bf);
    readln(bf, output_style);
    readln(bf);
    readln(bf, rec);
    readln(bf);
    readln(bf, ans);
    sexual := (ans = 'y');
    readln(bf);
    with MASC do
      begin
```

```
      readln(bf, sfunction);
      if sfunction = 4 then {glaciated}
        begin
          readln(bf);
          readln(bf, B)
        end
      else
        B := 1;
    end; {MASC}
   readln(bf);
   with FEMI do
    begin
      readln(bf, sfunction);
      if sfunction = 4 then {glaciated}
        begin
          readln(bf);
          readln(bf, B)
        end
      else
        B := 1;
    end;{FEMI}
   readln(bf);
   readln(bf, MASC.s);
   readln(bf);
   readln(bf, FEMI.s);
   readln(bf);
   readln(bf, MASC.nmig, MASC.nmigbarr);
   readln(bf);
   readln(bf, FEMI.nmig, FEMI.nmigbarr);

   readln(bf);
   readln(bf, ans);
   selfrandom := (ans = 'y');

   MASC.mig := MASC.nmig / MASC.ninds;
   FEMI.mig := FEMI.nmig / MASC.ninds;
 end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure set_listfile;
{-------------------------------------------------------------------}
 begin
   writeln(out_file, 'Run ', run, ' time ', TimeString, ' date ', DateString);
   writeln;
   writeln(out_file, 'runInt, Replicates, repNo, nsamples,twarm,tmax,dt,seedNo:');
   writeln(out_file, runInt);
   writeln(out_file, replicates);
   writeln(out_file, repNo);
   writeln(out_file, nsamples);
   writeln(out_file, twarm);
   writeln(out_file, tmax);
   writeln(out_file, dt);
   writeln(out_file, seedNo);
   writeln(out_file, 'ndemes,ninds,nstates,nloci,nbases,nmarkergenes,nchromosomes:');
   writeln(out_file, ndemes);
   writeln(out_file, MASC.ninds);
   writeln(out_file, nstates);
   writeln(out_file, nloci);
```

```
    writeln(out_file, nbases);
    writeln(out_file, nmarkergenes);
    writeln(out_file, nchromosomes);
    writeln(out_file, 'migration_style, basepop_style,output_style, rec:');
    writeln(out_file, migration_style);
    writeln(out_file, basepop_style);
    writeln(out_file, output_style);
    writeln(out_file, rec);
    writeln(out_file, 'sexual:', sexual);
    with MASC do
      begin
        writeln(out_file, 'Male pop variables');
        writeln(out_file, 'sfunction,Beta,s,nmig,nmigbarr,mig:');
        writeln(out_file, sfunction, ' ', B);
        writeln(out_file, s : 5 : 4);
        writeln(out_file, nmig);
        writeln(out_file, nmigbarr);
        writeln(out_file, mig);
      end;
    with FEMI do
      begin
        writeln(out_file, 'FEMale pop variables');
        writeln(out_file, 'sfunction,Beta,s,nmig,nmigbarr,mig:');
        writeln(out_file, sfunction, ' ', B);
        writeln(out_file, s : 5 : 4);
        writeln(out_file, nmig);
        writeln(out_file, nmigbarr);
        writeln(out_file, mig);
      end;
    writeln(out_file);
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure initialise;
{-------------------------------------------------------------------}
  begin
    nmidleft := (ndemes div 2);
    nmidright := (nmidleft + 1);
    RandSeed := seedNo; {SSSSSSSSSSSSSS}
    t := tstart;
    travel := 0;
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure initrep;
{-------------------------------------------------------------------}
    var
      i, count: longint;
      d, demeno, nIdenticalHaps: integer;
      sex: gender;
      sexvars: sexstuff;
      newsexpop: sexpopptrtype;
      Chameleon: diplotype;
  begin
    nIdenticalHaps := (2 * MASC.Ninds) div nstates;
```

```
for sex := male to female do
  begin
    if sex = male then
      begin
        sexvars := MASC;
        newsexpop := newpop.m
      end
    else
      begin
        sexvars := FEMI;
        newsexpop := newpop.f;
      end;
    with SEXVARS do
      begin
        case BASEPOP_STYLE of

          infinitedeme:
            begin
              if nmigbarr = 0 then { 1000 immigrant ( One ) individuals in an infinite native (
zero ) pop }
                begin
                  ninds := 1000;
                  for d := 1 to ndemes do
                    for i := 1 to ninds do
                      putdip(newsexpop, allOne, d, i){all genes set to one}
                end
            end;

          seperate_mixed_demes:
            for d := 1 to ndemes do
              begin
                if nstates = 2 then {Classic AllOne AlZero set up}
                  begin
                    for i := 1 to ninds do
                      putdip(newsexpop, allZero, d, i);
                    for i := 1 to ninds div 2 do
                      putdip(newsexpop, allOne, d, i);{half genes set to 1}
                  end
                else
                  begin { Pop is made up of nstates haplotypes, matrilines and patrilines NOT
set }
                    count := 0;
                    for i := 1 to ninds do
                      begin
                        Chameleon.ml := true;
                        Chameleon.pl := true;
                        INIT_STRAND(Chameleon.mt, (count div nIdenticalHaps) + 1);
                        count := count + 1;
                        INIT_STRAND(Chameleon.pt, (count div nIdenticalHaps) + 1);
                        count := count + 1;
                        putdip(newsexpop, Chameleon, d, i);
                      end;
                  end;
              end;
          rare_allele:
            for d := 1 to ndemes do
              begin
                for i := 1 to ninds div 10 do
                  putdip(newsexpop, allOne, d, i); {one}
                for i := 1 + ninds div 10 to (ninds div 10) * 2 do
                  putdip(newsexpop, RareOne, d, i);
```

```
                for i := (ninds div 10) * 2 + 1 to ninds do
                  putdip(newsexpop, AllZero, d, i);
              end;

            finite_deme_chain:
              begin
                if migration_style = waveofadvance then
                  begin
                    if nmigbarr = 0 then
                      for i := 1 to ninds do
                        putdip(newsexpop, allOne, 1, i){a colony of 1 indivs}
                    else { infinite colony on the left }
                      for i := 1 to ninds do
                        putdip(newsexpop, allZero, 1, i);{all genes set to zero}
                    for d := 2 to maxdemes do
                      for i := 1 to ninds do
                        putdip(newsexpop, allZero, d, i){all genes set to zero}
                  end
                else { normal stationary zone }
                  begin
                    for d := 1 to nmidleft do
                      for i := 1 to ninds do
                        putdip(newsexpop, allZero, d, i); {all genes set to zero}
                    for d := nmidright to ndemes do
                      for i := 1 to ninds do
                        putdip(newsexpop, allOne, d, i); {all genes set to one}
                  end;
              end;
          end;
        end;
    end;
  end;
{________________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure infinite_initrep;
{-------------------------------------------------------------------}
  begin
    FEMI.ninds := 0;
    MASC.ninds := 0; { all individuals are pure natives, and therefore can be disregarded }
    if paused then
      begin
        MASC.ninds := savedanindsM;
        FEMI.ninds := savedanindsF
      end;
  end;
{________________________________________________________________________}



{-------------------------------------------------------------------}
  procedure dumpDist (var sexvars: sexstuff;
                  sexpop: sexpopptrtype;
                  sex: gender;
                  do_cont: boolean);
{-------------------------------------------------------------------}
    var
      d, tally, SUM, SUMoSQR: LongInt;
      i, j: longint;
      mean: real;
```

```
{--------------------------------------------------------------------}
  {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
    procedure putinacont (blocksize: real);
  {--------------------------------------------------------------------}
      var
        i: integer;
    begin
      with sexvars do
        begin
          i := 1;
          while (blocksize <= conts[i].top) and (i < nbuckets) do{?}
            i := i + 1;
          conts[i].count := conts[i].count + 1;
        end;{?}
    end;
  {_______________________________________________________________}
{ block distribution }
{ conts distribution }
  begin
    with sexvars do
      begin
        for d := 1 to ndemes do
          begin
            SUM := 0;
            SUMoSQR := 0;
            for i := 1 to ninds do
              begin
                tally := bucketCOUNT_state(contributions, bucket, sexpop^[d]^[i].mt, output_style =
MYHYBRIDS);
                SUM := SUM + tally;
                SUMoSQR := SUMoSQR + sqr(tally);
                tally := bucketCOUNT_state(contributions, bucket, sexpop^[d]^[i].pt, output_style =
MYHYBRIDS);
                SUM := SUM + tally;
                SUMoSQR := SUMoSQR + sqr(tally);
              end;
            mean := SUM / ninds;
            if sex = male then
              write(out_mDist, mean : 10 : 8, '   ', (SUMoSQR / ninds) - sqr(mean) : 10 : 8, ' ')
            else
              write(out_fDist, mean : 10 : 8, '  ', (SUMoSQR / ninds) - sqr(mean) : 10 : 8, ' ');
            if sex = male then
              begin
                for i := 1 to nbuckets do
                  write(out_mDist, bucket[i].count : 10, '   ');
                writeln(out_mDist);
              end
            else
              begin
                for i := 1 to nbuckets do
                  write(out_fDist, bucket[i].count : 10, '  ');
                writeln(out_fDist);
              end;
            tally := 0;

            if do_cont then
              begin
                for i := 1 to nstates do {assuming constant population size}
                  begin
                    if contributions[i] > 0 then
                      tally := tally + 1;
```

```
                    putinacont(contributions[i]);
                  end;
                if sex = male then
                  begin
                    for i := 1 to nbuckets do
                      write(out_mHIbar, conts[i].count : 10, '   ');
                    write(out_mHIbar, tally, '   ');
                  end
                else
                  begin
                    for i := 1 to nbuckets do
                      write(out_fHIbar, conts[i].count : 10, '  ');
                    write(out_fHIbar, tally, '  ');
                  end;
                if tally < 50 then
                  begin
                    for i := 1 to nstates do
                      if contributions[i] > 0 then
                        if sex = male then
                          write(out_mHIbar, contributions[i] : 10 : 8, '    ')
                        else
                          write(out_fHIbar, contributions[i] : 10 : 8, '   ');

                  end;
                if sex = male then
                  writeln(OUT_mHIbar)
                else
                  writeln(OUT_fHIbar);
                zeroConts(contributions);
              end;
          end;
        emptybuckets(bucket, conts);
      end;
  end;
{-------------------------------------------------------------------}

{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure get_seed;   { sets new random seed }
{-------------------------------------------------------------------}
    var
      d: dateTimeRec;
  begin
    getTime(d);
    seedNo := (d.second * d.minute) + d.day
  end;
{___________________________________________________________________}


{"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure get_time;   { sets up time and date strings }
{-------------------------------------------------------------------}
    var
      d: dateTimeRec;
  begin
    getTime(d);
    DateString := stringof(d.day : 2, '/', d.month : 2, '/', d.year : 2);
    TimeString := stringof(d.hour : 2, ':', d.minute : 2, ':', d.second : 2);
  end;
{___________________________________________________________________}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure get_run;   { sets up run string}
{------------------------------------------------------------------------}
 begin
   reset(in_file, 'HP.inf');
   read(in_file, runInt);
   if paused then
     runInt := runInt - 1;
   Longint_to_string(runInt, run);
   rewrite(in_file);
   write(in_file, runInt + 1);
   close(in_file)
 end;
{______________________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure INITIALISE_ARRAYS;
{------------------------------------------------------------------------}
   var
     top: Longint;
     ok: boolean;
     totaldemes: integer;
{------------------------------------------------------------------------}
   procedure init_poparray (var p: sexpopptrtype);
     var
       d: integer;
   begin

     newmem(p, poparraylength, ok);
     if not ok then
       writeln('Uh oh, population not initialised');
     for d := 1 to totaldemes do
       begin
         newmem(p^[d], demearraylength, ok);
         if not ok then
           begin
             writeln('Uh oh, demearray ', d : 2, ' not initialised...ABORTING');
             goto 333;
           end;
       end;
   end;
{------------------------------------------------------------------------}

 begin
   top := mmAvail;
   if migration_style = waveofadvance then
     totaldemes := maxdemes
   else
     totaldemes := ndemes;
   demearraylength := MASC.ninds * sizeof(allOne);
     {array     :=[1..ninds] of diplotype     }

   poparraylength := totaldemes * length_PTR;
    {array     :=[1..ndemes] of demearrayptr}

   pvecarraylength := (MASC.ninds + 2) * length_REAL;
     {array     :=[0..maxinds+1] of        REAL}
```

```
    init_poparray(pop.m);
    init_poparray(pop.f);
    init_poparray(newpop.m);
    init_poparray(newpop.f);


    newmem(MASC.cumwptr, pvecarraylength, ok);
    newmem(FEMI.cumwptr, pvecarraylength, ok);
    if not ok then
      writeln('Uh oh, cumulative table not initialised');
    array_heap := top - mmAvail;
    writeln('Array heap ', array_heap);
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RETURN_ARRAYS;
{-----------------------------------------------------------------}
    procedure ret_poparray (var p: sexpopptrtype);
      var
        d, totaldemes: integer;
    begin
      if migration_style = waveofadvance then
        totaldemes := maxdemes
      else
        totaldemes := ndemes;

      for d := 0 to totaldemes - 1 do
        dispmem(p^[totaldemes - d], demearraylength);
      dispmem(p, poparraylength);
    end;
{-----------------------------------------------------------------}
  begin
    dispmem(FEMI.cumwptr, pvecarraylength);
    dispmem(MASC.cumwptr, pvecarraylength);
    ret_poparray(newpop.f);
    ret_poparray(newpop.m);
    ret_poparray(pop.f);
    ret_poparray(pop.m);
  end;
{_______________________________________________________________}


{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure get_context;
{-----------------------------------------------------------------}
    var
      d, i, tally: integer;
      state: string;
{-----------------------------------------------------------------}
  begin
    reset(context, 'h.con');
    readln(context, state);
    paused := state <> 'finished';
    if paused then
      begin
```

```
        tally := 0;
        readln(context, batch_fname);
        readln(context, completeruns);
        readln(context, pausedthisrun);
        readln(context, savednindsM);
        readln(context, savednindsF);
        readln(context, tStart);
        if state = 'processing' then
{ state=processing: machine has crashed, or rude interruption. }
        {Population data will be corrupted, therefore }
          begin
            tStart := 0;
            SavednindsM := 0;
            SavednindsF := 0;
            pausedthisrun := 0;
          end;
        close(context);
        rewrite(context, 'h.con');
        writeln(context, 'processing');
        writeln(context, batch_fname);
        writeln(context, completeruns);
        writeln(context, pausedthisrun);
        writeln(context, savednindsM);
        writeln(context, savednindsF);
        writeln(context, tStart);

        reset(bf, batch_fname);
        repeat
          readln(bf, control_string);
          askstuff_batch;
          tally := tally + replicates
        until tally > completeruns;
        repStart := replicates - (tally - completeruns - 1);
        t := tstart;
      end;
    close(context);
  end;
{-----------------------------------------------------------------}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure RETURN_AFTER_BREAK;
{-----------------------------------------------------------------}
    var
      i, j: longint;
      d: integer;
      sex: gender;
{-----------------------------------------------------------------}
  begin
    RESET_SECRET;
    for sex := male to female do
      for d := 1 to ndemes do
        if sex = male then
          for i := 1 to savednindsM do
            loaddip(newpop.m, d, i)
        else
          for i := 1 to savednindsF do
            loaddip(newpop.f, d, i);
    CLOSE_SECRET;
    LOAD_NEWPOOL;
  end;
{-----------------------------------------------------------------}
```

```
{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure save_context;
{-----------------------------------------------------------------------}
   var
     d, tally: integer;
     i, j: longint;
     sex: gender;
{-----------------------------------------------------------------------}
 begin
   for i := 1 to 20 do
     writeln('SAVING DATA IN PROGRESS, PLEASE WAIT..');
   REWRITE_SECRET('h.oz');
   for sex := male to female do
     for d := 1 to ndemes do
       if sex = male then
         for i := 1 to MASC.ninds do
           savedip(newpop.m, d, i)
       else
         for i := 1 to FEMI.ninds do
           savedip(newpop.f, d, i);
   SAVE_NEWPOOL('h.dls');
   rewrite(context, 'h.con');
   writeln(context, 'cleanly paused'); { this state can only be reached at this point }
   writeln(context, batch_fname);
   writeln(context, completeruns);
   writeln(context, pausedthisrun + 1);
   writeln(context, MASC.ninds);
   writeln(context, FEMI.ninds);
   writeln(context, t);
   close(context);
   for i := 1 to 20 do
     writeln;
   writeln('Done, thankyou.');
 end;
{-----------------------------------------------------------------------}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
 procedure update_context;
{-----------------------------------------------------------------------}
 begin
   rewrite(context, 'h.con');
   writeln(context, 'processing');
   writeln(context, batch_fname);
   writeln(context, completeruns);
   writeln(context, pausedthisrun);
   writeln(context, savednindsM);
   writeln(context, savednindsF);
   writeln(context, tStart);
   close(context);
 end;
{-----------------------------------------------------------------------}

{-----------------------------------------------------------------------}
 procedure check_for_breaks;
{-----------------------------------------------------------------------}
   var
     temp: boolean;
 begin
   getmouse(newpoint);
```

```
    temp := breakoff;
    if not breakoff then
      breakoff := button
    else if button then
      breakoff := false;
    if temp <> breakoff then
      writeln('Breaking = ', not breakoff);

    if breakoff then
      oldpoint := newpoint;

    if ((newpoint.v > oldpoint.v + 3) or (newpoint.v < oldpoint.v - 3)) or ((newpoint.h > oldpoint.h
+ 3) or (newpoint.h < oldpoint.h - 3)) then
      begin
        save_context;
        goto 333;
      end;

    oldpoint := newpoint;
    if crash or explode then
      goto 999;
  end;
{--------------------------------------------------------------------}
{--------------------------------------------------------------------}
  procedure INIT_OUTPUT;
{--------------------------------------------------------------------}
    var
      hety, hindex, fit, freqs, conts, dists, UBmarkers: boolean;
      i: integer;
  begin
    hety := false;
    hindex := false;
    fit := false;
    freqs := false;
    conts := false;
    dists := false;
    UBmarkers := false;
    rewrite(out_file, concat(path, ' Info '));
    rewrite(out_Mgen, concat(path, ' MGen'));
    rewrite(out_Fgen, concat(path, ' FGen'));
    case OUTPUT_STYLE of
      ANCESTRY:
        begin
          conts := true;
          dists := true;
        end;
      MYHYBRIDS:
        begin
          UBmarkers := false;
          dists := false;
          hindex := true;
          hety := true;
        end;
      indices:
        begin
          hety := true;
          hindex := true;
        end;
    end;
    if freqs then
```

```
    begin
      out_Mpbname := concat(path, '  mPbar');
      out_fpbname := concat(path, '  fPbar');
      rewrite(out_mPbar, out_mpbname);
      rewrite(out_fPbar, out_fpbname);
    end;
  if fit then
    begin
      out_mWBname := concat(path, ' mWbar');
      out_mWVname := concat(path, ' mWvar');
      out_fWBname := concat(path, ' fWbar');
      out_fWVname := concat(path, ' fWvar');
      rewrite(out_mWbar, out_mwbname);
      rewrite(out_mWvar, out_mwvname);
      rewrite(out_fWbar, out_fwbname);
      rewrite(out_fWvar, out_fwvname);
    end;

  if hindex then
    begin
      if Dwave then
        out_mHIvname := concat(path, ' mDwave')
      else
        out_mHIvname := concat(path, ' mmarker2');
      out_mHIbname := concat(path, ' mHIbar');
      if Dwave then
        out_fHIvname := concat(path, ' fDwave')
      else
        out_fHIvname := concat(path, ' fMarker2');
      out_fHIbname := concat(path, ' fHIbar');
      rewrite(out_mHIvar, out_mHIvname);
      rewrite(out_mHIbar, out_mHIbname);
      rewrite(out_fHIvar, out_fHIvname);
      rewrite(out_fHIbar, out_fHIbname);
      if nmarkergenes > 1 then
        begin
          rewrite(out_mWbar, concat(path, ' mDiseq'));
          rewrite(out_fWbar, concat(path, ' fDiseq'));
        end;
      rewrite(out_mDIST, concat(path, ' mMarker1'));
      rewrite(out_fDIST, concat(path, ' fMarker1'));
      if migration_style = waveofadvance then
        for i := 1 to maxdemes do
          begin
            write(out_mDIST, 0, '  ');
            write(out_fDIST, 0, '    ');
          end;
      writeln(out_mDIST);
      writeln(out_fDIST);
      write(out_mDIST, '-10     ');
      write(out_fDIST, '-10    ');
      if UBmarkers then
        begin
          rewrite(out_mPbar, concat(path, ' mPtDNA'));
          rewrite(out_fPbar, concat(path, ' fPtDNA'));
          rewrite(out_mWbar, concat(path, ' mMtDNA'));
          rewrite(out_fWbar, concat(path, ' fMtDNA'));
          for i := 1 to maxdemes do
            begin
              write(out_mPbar, 0, '    ');
              write(out_fPbar, 0, '  ');
```

```
          write(out_mWbar, 0, '   ');
          write(out_fWbar, 0, '    ');
        end;
      writeln(out_mPbar);
      writeln(out_fPbar);
      write(out_mPbar, '-10   ');
      write(out_fPbar, '-10  ');
    end;
  writeln(out_mWbar);
  writeln(out_fWbar);
  write(out_mWbar, '-10  ');
  write(out_fWbar, '-10    ');
  if migration_style = waveofadvance then
    begin
      for i := 1 to maxdemes do
        begin
          write(out_mHIbar, 0, '   ');
          write(out_fHIbar, 0, '    ');
          write(out_mHIvar, 0, '  ');
          write(out_fHIvar, 0, '    ');
        end;
    end;
  writeln(out_mHIbar);
  writeln(out_fHIbar);
  writeln(out_mHIvar);
  writeln(out_fHIvar);
  write(out_mHIbar, '-10     ');
  write(out_fHIbar, '-10    ');
  write(out_mHIvar, '-10     ');
  write(out_fHIvar, '-10    ');
 end;
if conts then
 begin
  out_mHIbname := concat(path, ' mConts');
  out_fHIbname := concat(path, ' fConts');
  rewrite(out_mHIbar, out_mHIbname);
  rewrite(out_fHIbar, out_fHIbname);
 end;
if dists then
 begin
  out_mDistname := concat(path, ' mDists');
  out_fDistname := concat(path, ' fDists');
  rewrite(out_mdist, out_mDistname);
  rewrite(out_fdist, out_fDistname);
 end;
if hety then
 begin
  out_fHTbname := concat(path, ' fHTbar');
  out_mHTbname := concat(path, ' mHTbar');
  rewrite(out_mHTbar, out_mHTbname);
  rewrite(out_fHTbar, out_fHTbname);
  out_fHTvname := concat(path, ' fSpare');
  out_mHTvname := concat(path, ' mSpare');
  rewrite(out_mHTvar, out_mHTvname);
  rewrite(out_fHTvar, out_fHTvname);
  if migration_style = waveofadvance then
    begin
      for i := 1 to maxdemes do
        begin
          write(out_mHTbar, 0, '    ');
          write(out_fHTbar, 0, '  ');
```

```
                write(out_mHTvar, 0, '    ');
                write(out_fHTvar, 0, '   ');
            end;
        end;
      writeln(out_mHTbar);
      writeln(out_fHTbar);
      write(out_mHTbar, '-10    ');
      write(out_fHTbar, '-10   ');
      writeln(out_mHTvar);
      writeln(out_fHTvar);
      write(out_mHTvar, '-10    ');
      write(out_fHTvar, '-10   ');
    end;
  end;
{-----------------------------------------------------------------------}

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
  procedure DO_OUTPUT;
{-----------------------------------------------------------------------}
{____________________________________________________________________________}
{For the two populations/sexes this updates SOME from the list of statistics: }

{wbar, wvar: Average and variance in fitness }
{HIbar, HIvar:Average and variance in hybrid index (0<i<1) }
{Mtbar,Ptbar:Average and variance in materally and paternally inherited factors }
{HTbar,HTvar:Average and variance in heterozygosity }

{Blocks: Distribution of block sizes }
{Conts: Distribution of contributions of genetic material }

{collected every dt generations}
    var
      d: demetype;
      g: genetype;
      wb, vw, htb, vht, hib, vhi, mt, pt: real;
      gv, pbar, time, diseq, p1, p2, pHT: real;
      onetot, zerotot, i: integer;
      UBmarkers, hety: boolean;
  begin
    UBmarkers := false;
    hety := true;
    if (t >= twarm) and ((t mod dt = 0) or (t <= 40)) then
      begin
        sample := sample + 1;
        write('Updating stats for generation = ', t : 2, ' ');
        time := t;
        distribution := t > (tmax - 4 * dt);
        write(out_fGen, time : 3 : 1, '   '); {NOW t IS ONLY WRITTEN TO THE GENERAL FILE
}
        if migration_style = waveofadvance then
          begin
            write(out_mGen, travel, '    ');
            travel := 0
          end;
        if basepop_style = infinitedeme then
          begin
            write(out_mGen, MASC.NINDS, '    ');
            write(out_fGen, FEMI.NINDS, ' ');
          end;
        writeln(out_fGen);
        writeln(out_mGen);
```

```
for d := 1 to ndemes do
  begin
    with MASC do
      begin
        case output_style of
          boring:
            begin
              find_wbar(s, B, sfunction, ninds, nmarkergenes, nchromosomes, pop.m, d, wb,
vw, htb, vht, hib, vhi, true);
              genefreqstats(ninds, nmarkergenes, pop.m, d, gv, pbar);
        {tv := total_variance(ninds, nmarkergenes, pop.m, d);}

              write(out_mHTbar, HtB : 5 : 3, '    ');
              write(out_mHTvar, vht : 5 : 3, '  ');
              if nstates = 2 then
                begin
                  write(out_mHIbar, HIB : 5 : 3, '  ');
                  write(out_mHIvar, vhi : 5 : 3, '    ');
                end;
              for g := 1 to nmarkergenes do
                write(out_mPbar, getgenefreq(ninds, pop.m^[d], g, 1) : 5 : 3, '    ');


              write(out_mWbar, wb : 5 : 3, '    ');
              write(out_mWvar, vw : 5 : 3, '    ');
            end;
          MYHYBRIDS:
            begin
              Diseq := getgenediseq(ninds, pop.m^[d], 1, 2, p1, p2, pHT, 1);
              if Dwave then
                begin
                  write(out_mHIvar, Diseq : 5 : 3, '    ')
                end
              else
                begin
                  write(out_mHIvar, p1 : 5 : 3, '  ');
                  write(out_mDIST, p2 : 5 : 3, '  ');
                  write(out_mWbar, Diseq : 5 : 3, '    ')
                end;
              if UBmarkers then
                begin
                  find_extranuclear(ninds, pop.m, d, mt, pt);
                  write(out_mPbar, pt : 5 : 3, '    ');
                  write(out_mWbar, mt : 5 : 3, '  ');
                end;
              find_wbar(s, B, sfunction, ninds, nmarkergenes, nchromosomes, pop.m, d, wb,
vw, htb, vht, hib, vhi, true);
              write(out_mHIbar, HIB : 5 : 3, '  ');
              if hety then
                begin
                  write(out_mHTbar, HtB : 5 : 3, '    ');
                  write(out_mHTvar, pHt : 5 : 3, '  ');
                end;
            end;
        end;
      end;
    with FEMI do
      case output_style of
        MYHYBRIDS:
          begin
            Diseq := getgenediseq(ninds, pop.f^[d], 1, 2, p1, p2, pHT, 1);
```

```
                    if Dwave then
                      write(out_fHIvar, Diseq : 5 : 3, '  ')
                    else
                      begin
                        write(out_fWbar, Diseq : 5 : 3, '   ');
                        write(out_fHIvar, p1 : 5 : 3, '    ');
                        write(out_fDIST, p2 : 5 : 3, '    ');
                      end;

                    if UBmarkers then
                      begin
                        find_extranuclear(ninds, pop.f, d, mt, pt);
                        write(out_fPbar, pt : 5 : 3, '   ');
                        write(out_fWbar, mt : 5 : 3, '    ');
                      end;
                    find_wbar(s, B, sfunction, ninds, nmarkergenes, nchromosomes, pop.f, d, wb,
vw, htb, vht, hib, vhi, true);
                    write(out_fHIbar, HIB : 5 : 3, '     ');
                    if hety then
                      begin
                        write(out_fHTbar, HtB : 5 : 3, '    ');
                        write(out_fHTvar, pHt : 5 : 3, '    ');
                      end;
                  end;
              end;
          end; { deme-by-deme output }
        case output_style of
          indices:
            begin
              writeln(out_mHTvar);
              writeln(out_mHTbar);
              writeln(out_fHTvar);
              writeln(out_fHTbar);
              writeln(out_mHIvar);
              writeln(out_mHIbar);
              writeln(out_fHIvar);
              writeln(out_fHIbar);
            end;
          MYHYBRIDS:
            begin

{    dumpDist(MASC, pop.m, male, false);}
{    dumpDist(FEMI, pop.f, female, false);}
              writeln(out_mDIST);
              write(out_mDIST, '-10     ');
              writeln(out_fDIST);
              write(out_fDIST, '-10    ');

              writeln(out_mHIbar);
              write(out_mHIbar, '-10     ');
              writeln(out_mHIvar);
              write(out_mHIvar, '-10     ');
              writeln(out_fHIbar);
              write(out_fHIbar, '-10    ');
              writeln(out_fHIvar);
              write(out_fHIvar, '-10    ');
              if hety then
                begin
                  writeln(out_mHTbar);
                  write(out_mHTbar, '-10    ');
                  writeln(out_fHTbar);
```

```
                write(out_fHTbar, '-10  ');
                writeln(out_mHTvar);
                write(out_mHTvar, '-10   ');
                writeln(out_fHTvar);
                write(out_fHTvar, '-10  ');
              end;
            if UBmarkers then
              begin
                writeln(out_mPbar);
                write(out_mPbar, '-10   ');
                writeln(out_fPbar);
                write(out_fPbar, '-10  ');
              end;
            writeln(out_fWbar);
            write(out_fWbar, '-10    ');
            writeln(out_mWbar);
            write(out_mWbar, '-10  ');
          end;
        ANCESTRY:
          begin
            dumpDist(MASC, pop.m, male, true);
            dumpDist(FEMI, pop.f, female, true);
          end;
      end;
    end;
    get_time;
    writeln('Time ', timeString, ' ', dateString);
  end;

{___________________________________________________________}

{-------------------------------------------------------}
  procedure CLOSE_OUTPUT;
{-------------------------------------------------------}
    var
      UBmarkers, hety: boolean;
  begin
    UBmarkers := false;
    hety := true;
    close(out_file);
    close(out_Mgen);
    close(out_Fgen);
    case output_style of
      MYHYBRIDS:
        begin
          writeln(out_mHIvar);
          writeln(out_mHIvar, '-20     ');
          close(out_mHIvar);
          writeln(out_mHIbar);
          writeln(out_mHIbar, '-20     ');
          close(out_mHIbar);
          writeln(out_fHIvar);
          writeln(out_fHIvar, '-20    ');
          close(out_fHIvar);
          writeln(out_fHIbar);
          writeln(out_fHIbar, '-20    ');
          close(out_fHIbar);
          writeln(out_mDist);
          writeln(out_mDist, '-20    ');
          close(out_mDist);
          writeln(out_fDist);
```

```
          writeln(out_fDist, '-20  ');
          close(out_fDist);
          if hety then
            begin
              writeln(out_mHtBar);
              writeln(out_mHtBar, '-20    ');
              close(out_mHtBar);
              writeln(out_fHtBar);
              writeln(out_fHtBar, '-20   ');
              close(out_fHtBar);
              writeln(out_mHtvar);
              writeln(out_mHtvar, '-20    ');
              close(out_mHtvar);
              writeln(out_fHtvar);
              writeln(out_fHtvar, '-20   ');
              close(out_fHtvar);
            end;
          if UBmarkers then
            begin
              writeln(out_mPbar);
              writeln(out_mPbar, '-20   ');
              close(out_mPbar);
              writeln(out_fPbar);
              writeln(out_fPbar, '-20 ');
              close(out_fPbar);
            end;
          writeln(out_mWbar);
          writeln(out_mWbar, '-20 ');
          close(out_mWbar);
          writeln(out_fWbar);
          writeln(out_fWbar, '-20    ');
          close(out_fWbar);
        end;
      ANCESTRY:
        begin
          close(out_mHIbar);
          close(out_fHIbar);
          close(out_mDist);
          close(out_fDist);
        end;
    end;
  end;
{-----------------------------------------------------------------}
{-----------------------------------------------------------------}
  procedure MIGRATION;{[ RTN 3 p8]}
{-----------------------------------------------------------------}
  begin
{as the old population is now unnecessary, we can reclaim all its storage}
{I haven't bothered creating "dummies": I just have no migration beyond the ends}
    case Migration_style of
      no_migration:
        begin
        end;
      normal:
        begin
          with MASC do
            migrate(pop.m, newpop.m, nmig, nmigbarr, ndemes, ninds, nmidleft, nmidright);
          with FEMI do
            migrate(pop.f, newpop.f, nmig, nmigbarr, ndemes, ninds, nmidleft, nmidright);
          swappop(pop, newpop);
        end;
```

```
          waveofadvance:
            begin
              with MASC do
                advance(pop.m, newpop.m, nmig, nmigbarr, ndemes, ninds, nmidleft, nmidright);
              with FEMI do
                advance(pop.f, newpop.f, nmig, nmigbarr, ndemes, ninds, nmidleft, nmidright);
              swappop(pop, newpop);
            end;
          from_infinite:
            begin
              case basepop_style of
                infinitedeme:
                  begin
                    with MASC do
                      infinite_immigrate(pop.m, nmig, nmigbarr, ndemes, allONE, allZero, ninds);
                    with FEMI do
                      infinite_immigrate(pop.f, nmig, nmigbarr, ndemes, allONE, allZero, ninds)
                  end;
                finite_deme_chain:
                  begin
                    with MASC do
                      immigrate(pop.m, nmigbarr, ndemes, allONE);
                    with FEMI do
                      immigrate(pop.f, nmigbarr, ndemes, allONE)
                  end;
              end;
            end;
          double_from_infinite:
            begin
              with MASC do
                double_immigrate(pop.m, nmig, nmigbarr, ndemes, allONE, allZero);
              with FEMI do
                double_immigrate(pop.f, nmig, nmigbarr, ndemes, allONE, allZero)
            end;
        end;
      end;
    {-------------------------------------------------------------------}

    {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
    {""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
    {""""""""""""[ RTN 1 p6]"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}
begin
  completeruns := 0;
  pausedthisrun := 0;
  repstart := 1;
  tstart := 0;
  breakoff := false;
  get_context;

  windowrect.topleft.v := 50;
  windowrect.topleft.h := 0;
  windowrect.botright.v := 380;
  windowrect.botright.h := 510;
  SetTextRect(windowRect);
  ShowText;
  installmem(paused, done);
  if not done then
    goto 333;
  INIT_STRAND(allOne.mt, 1);
  INIT_STRAND(allOne.pt, 1);
  allOne.ml := true;
```

```
  allOne.pl := true;
  INIT_STRAND(allZero.mt, 0);
  INIT_STRAND(allZero.pt, 0);
  allZero.ml := false;
  allZero.pl := false;
  INIT_STRAND(allHet.mt, 0);
  INIT_STRAND(allHet.pt, 1);
  allZero.ml := false;
  allZero.pl := true;
  INIT_STRAND(RareOne.mt, 0);
  INIT_STRAND(RareOne.pt, 0);
  RareOne.ml := false;
  RareOne.pl := false;
  write('Do you want to read parameters from a batch file ? ');
  if paused then
    begin
      batch := true;
      writeln('Auto reading....');
    end
  else
    begin
      readln(ans);
      batch := (ans = 'y');
      if batch then
        begin
          batch_fname := OldFileName('Select Batch File');
          reset(bf, batch_fname);
          readln(bf, control_string)
        end
      else
        control_string := 'go_ahead';
    end;
  new(name);
  name^ := harddisc;
  error := getvol(name, vol);
  creator := 'SJEB';
  tipe := 'Jntn';


{----------------------------------------------------------------}
{        Batch loop                                              }
{----------------------------------------------------------------}
  while control_string = 'go_ahead' do
    begin
      if not paused then
        askstuff_batch;
      if basepop_style = infinitedeme then
        begin
          MASC.ninds := infinitedememax;
          FEMI.ninds := infinitedememax; { for array initialisation }
        end;
      nsamples := ((tmax - twarm) div dt) + 1;
     {rec defines a Poisson distribution of recombination events }
{    mu := (-1 * ln(1 - 2 * rec)) * 0.5; }
     { Haldane's map function }
     {halved as we are only considering two of four chromatids   }
{ changed as of 7/8/92. We now input the total map length}
      mu := rec;
      writeln('Available memory is ', mmAvail : 10);
      INITIALISE_ARRAYS;
      origndemes := ndemes;
```

```
    INITIALISE_POSITIONS(nloci, nmarkergenes, nchromosomes);{positions of marker genes}
    writeln('Available memory is ', mmAvail : 10);

    pool_heap := INITIALISE_POOLS(mmAvail);
    writeln('Available memory is ', mmAvail : 10);
    with MASC do
      setbuckets(bucket, conts, nstates);
    with FEMI do
      setbuckets(bucket, conts, nstates);
    getmouse(oldpoint);

{-----------------------------------------------------------------}
{       Replicate loop          [RTN 2 p6]                         }
{-----------------------------------------------------------------}

    path := concat(harddisc, ':', replicatesummary);
    error := dircreate(vol, unknownint, path, unknownLongint);
    for repNo := repstart to replicates do
      begin
        if migration_style = waveofadvance then
          ndemes := origndemes;
        explode := false;
        get_run;
        get_time;
        writeln('Run ', run, '. Replicate', repNo, ' of', replicates);
        if batch then
          begin
            if selfrandom then
              get_seed
            else
              readln(bf, seedNo);
            writeln('Random seed: ', seedNo)
          end;
        path := concat(harddisc, ':', replicatesummary);
        if paused then
          path := concat(path, ':R', run, stringof(pausedthisrun))
        else
          path := concat(path, ':R', run);
        error := dircreate(vol, unknownint, path, unknownLongint);
        path := concat(path, ':R', run);
        INIT_OUTPUT;
        initialise;
        nfix := 0;
        set_listfile;

        SWAP_POOLS(crash);
{   if basepop_style = rare_allele then }
{   begin }
{   FORCE_JUNCTION(RareOne.mt, 1.5); }
{   FORCE_JUNCTION(RareOne.pt, 1.5); }
{   end; }

        if basepop_style = infinitedeme then
          infinite_initrep
        else
          initrep;

        tstart := 0;

        if paused then
          RETURN_AFTER_BREAK;
```

```
        paused := false;
        swappop(pop, newpop);

{--------------------------------------------------------------}
{       Generation loop                                         }
{--------------------------------------------------------------}
        repeat
          DO_OUTPUT;

          MIGRATION;
{[ RTN 4 p9]}
          case basepop_style of
            infinitedeme:
              infinite_reproduce(pop, newpop, rec);
            otherwise
              reproduce(pop, newpop, rec);
          end;

          t := t + 1;

          if standalone then
            CHECK_FOR_BREAKS;

          SWAP_POOLS(crash);
          swappop(pop, newpop);
          if migration_style = waveofadvance then
            begin
              writeln('Ndemes=', ndemes);
              temp1 := ndemes;
              temp2 := ndemes;
              with MASC do
                adjustdemes(pop.m, newpop.m, temp1, ninds, travel, stretch, allZero);
              with FEMI do
                adjustdemes(pop.f, newpop.f, temp2, ninds, travel, stretch, allZero);
              if temp1 > temp2 then
                ndemes := temp1
              else
                ndemes := temp2;
              swappop(pop, newpop);
            end;
        until t >= tmax;
{--------------------------------------------------------------}
{       End of generation loop                                  }
{--------------------------------------------------------------}

        DO_OUTPUT;

999:
        if crash then
          begin
            writeln(out_file, 'Out of Memory.. run terminated on run', run);
            writeln('OUT OF MEMORY... This run is terminated.')
          end;
        if explode then
          begin
            writeln(out_file, 'Population explosion.. run terminated on run', run);
            writeln('POPULATION EXPLOSION... This run is terminated.')
          end;
        CLOSE_OUTPUT;
        pausedthisrun := 0;
        completeruns := completeruns + 1;
```

```
          UPDATE_CONTEXT;
        end;
{------------------------------------------------------------}
{       End of Replicate loop                                 }
{------------------------------------------------------------}
      repstart := 1;
      RETURN_POOLS;
      RETURN_ARRAYS;
      writeln('Memory returned, ', mmAvail, ' bytes available.');
      if batch then
        readln(bf, control_string)
      else
        control_string := 'finished';
    end;
{------------------------------------------------------------}
{       End of Batch loop                                     }
{------------------------------------------------------------}
  rewrite(context, 'h.con');
  writeln(context, 'finished');
333:

end.
```